Requested Patent: JP6326813A

Title: FAULT RECOVERY CONTROL SYSTEM ;

Abstracted Patent: JP6326813 ;

Publication Date: 1994-11-25 ;

Inventor(s): TANAKA SHINICHI ;

Applicant(s): OKI ELECTRIC IND CO LTD ;

Application Number: JP19930134006 19930512 ;

Priority Number(s): ;

IPC Classification: H04N1/00; G06F13/00; G06F15/30 ;

Equivalents: JP2951151B2 ;

ABSTRACT:

PURPOSE:To avoid useless re-transmission from a data sender and to attain the processing corresponding to the occurrence of a fault quickly even by the data sender.

CONSTITUTION:Data from facsimile equipments 11, 21, 31 being plural data senders are sent to an area center 40 being a management center, where the data are processed. An FCU 410 of the area center 40 is provided with a recovery pattern table 101 in which a pattern of recovery processing corresponding to a fault occurrence time is set in advance. On the occurrence of a fault in an MCU 42, a recovery processing section 102 provided in the FCU 410 references to the recovery pattern table 101 to execute the recovery processing corresponding to the fault occurrence time.

(19)日本国特許庁（ＪＰ）　(12)公開特許公報(A)　(11)特許出願公開番号

特開平6－326813

(43)公開日　平成６年(1994)11月25日

(51)Int.Cl.[5]		識別記号	庁内整理番号	ＦＩ	技術表示箇所
Ｈ０４Ｎ	1/00	１０６	Ｃ　7232－5Ｃ		
Ｇ０６Ｆ	13/00	３０１	Ｋ		
	15/30		Ｍ		
		３１０			
// Ｇ０７Ｄ	9/00	４３６	Ｂ　8111－3Ｅ		

審査請求　未請求　請求項の数４　ＦＤ　(全 17 頁)

(21)出願番号　特願平5－134006

(22)出願日　平成５年(1993)５月12日

(71)出願人　000000295
沖電気工業株式会社
東京都港区虎ノ門１丁目７番12号

(72)発明者　田中　信一
東京都港区虎ノ門１丁目７番12号　沖電気工業株式会社内

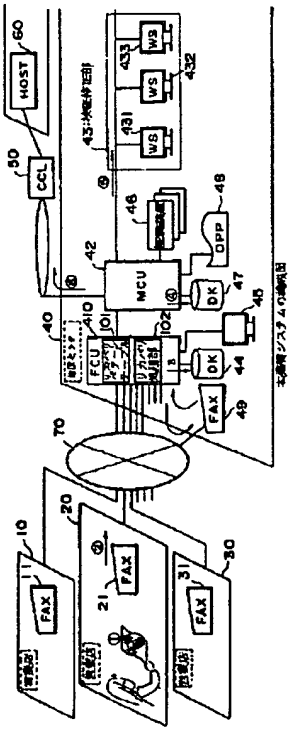
(74)代理人　弁理士　佐藤　幸男

(54)【発明の名称】　障害リカバリ制御システム

(57)【要約】

【構成】　複数のデータ送信元であるファクシミリ１１、２１、３１からのデータは、管理センタとしての地区センタ４０に送信され、地区センタ４０でこれらのデータ処理が行われる。地区センタ４０のＦＣＵ４１０には、障害発生時刻に対応したリカバリ処理のパターンが予め設定されているリカバリパターンテーブル１０１が設けられている。ＭＣＵ４２で障害が発生した場合、ＦＣＵ４１０に設けられたリカバリ処理部１０２は、リカバリパターンテーブル１０１を参照し、障害発生時刻に対応したリカバリ処理を行う。

【効果】　データ送信元からの無駄な再送をなくすことができると共に、データ送信元でも速やかに障害発生に対応した処理を行うことができる。



1

【特許請求の範囲】

【請求項1】　複数のデータ送信元と、前記複数のデータ送信元と通信回線を介して接続され、各々のデータ送信元からのデータを蓄積し、データ処理を行う管理センタとからなるシステムにおいて、

前記管理センタは、

当該管理センタでの障害発生を想定し、その障害発生時刻に対応したリカバリ処理のパターンを予め設定したリカバリパターンテーブルと、

前記管理センタで障害が発生した場合、前記リカバリパターンテーブルを参照し、障害発生時刻に対応したリカバリ処理を行うリカバリ処理部とを備えたことを特徴とする障害リカバリ制御システム。

【請求項2】　リカバリパターンテーブルは、管理センタでの障害発生を想定し、その障害発生時刻と、データ送信元からのデータ送信先とに対応してリカバリ処理のパターンを予め設定したことを特徴とする請求項1記載の障害リカバリ制御システム。

【請求項3】　リカバリ処理部は、障害発生後にデータ受信を再開するか否かを制御する受信制御と、障害発生までの未処理データをデータ送信元に返送するか否かを制御する返送制御とに基づくパターンで動作することを特徴とする請求項1または2記載の障害リカバリ制御システム。

【請求項4】　リカバリ処理部は、障害発生後にデータ受信を再開するか否かを制御する受信制御と、障害発生までの未処理データをデータ送信元に返送するか否かを制御する返送制御とに基づきパターンを決定すると共に、複数のデータ送信元に対して各パターンに対応した同報メッセージを出力することを特徴とする請求項1または2記載の障害リカバリ制御システム。

【発明の詳細な説明】

【0001】

【産業上の利用分野】本発明は、例えば、金融機関の営業店と地区センタとの間で用いられるＦＡＸ－ＯＣＲシステムでの障害発生時のリカバリを行う障害リカバリ制御システムに関する。

【0002】

【従来の技術】銀行等の金融機関では、各営業店から地区センタに振替依頼書のイメージデータをファクシミリで送信し、地区センタが各営業店からのデータ処理を行ってホストに転送するといったデータ処理システム、所謂ＦＡＸ－ＯＣＲシステムが採用されている。そして、このようなデータ処理システムを採用することによって、営業店の負担を軽減し、業務の能率向上を図っている。

【0003】図2は、このようなＦＡＸ－ＯＣＲシステムの説明図である。図のシステムは、営業店１０、２０、３０と、地区センタ４０と、通信制御装置（ＣＣＬ）５０と、ホスト６０と、通信回線７０とからなる。

2

営業店１０、２０、３０には、それぞれファクシミリ（ＦＡＸ）１１、２１、３１が備えられている。これらのファクシミリ１１、２１、３１は、通信回線７０を介して地区センタ４０のファクシミリ受信制御部（ＦＣＵ：ファクシミリコントロールユニット）４１に接続されている。

【0004】地区センタ４０は、ファクシミリ受信制御部（以下、ＦＣＵという）４１と、制御部（ＭＣＵ：メッセージコントロールユニット）４２、検証修正部４３等からなる。ＦＣＵ４１は、ファクシミリデータの送受信を行うためのものであり、ディスク装置（ＤＫ）４４と、ワークステーション４５とが接続されている。ディスク装置４４は、ファクシミリ送受信されるデータ等を一時的に格納するための外部記憶装置である。また、ワークステーション４５は、制御部やディスプレイおよびキーボード等からなり、ファクシミリ送受信されるデータを表示し、操作するためのものである。

【0005】制御部（以下、ＭＣＵという）４２は、認識装置４６、ディスク装置（ＤＫ）４７、プリンタ（ＯＰＰ）４８および検証修正部４３を接続し、ＦＣＵ４１からのデータを受信して、認識装置４６に対して文字認識処理を指示したり、検証修正部４３での出力データをホスト６０に対して送信するといった地区センタ４０における各種の処理を管理する機能を有している。

【0006】認識装置４６は、ＦＣＵ４１が受信したイメージデータに含まれる帳票上の各フィールドの文字を認識し、コードデータを得るための文字認識部である。また、ディスク装置４７は、認識装置４６で文字認識されたデータを一時格納するための外部記憶装置であり、プリンタ４８は、文字認識結果等を印刷出力したい場合等に使用されるものである。

【0007】検証修正部４３は、複数のワークステーション４３１～４３３からなる。ワークステーション４３１～４３３は、ディスプレイやキーボード等を備えたパーソナルコンピュータで構成されて、各々オペレータが配属され、上記のイメージータとコードデータとを照合してその認識結果が正しいか否かの検証と、誤りがあった場合はその修正を行うための装置でる。更に、ファクシミリ（ＦＡＸ）４９は、地区センタ４０内に設けられたファクシミリで、通信回線７０あるいは地区センタ４０内のＰＢＸ（構内交換機）を介して営業店１０～３０やホスト６０等とのデータの授受を行うものである。

【0008】通信制御装置５０は、地区センタ４０のＭＣＵ４２からの送受信制御を行う装置であり、ホスト（ＨＯＳＴ）６０は、営業店１０～３０や地区センタ４０を統轄するセンタである。また、通信回線７０は、公衆回線や専用回線で構成され、営業店１０～３０と地区センタ４０また、営業店１０～３０とホスト６０との通信接続を行うネットワークである。

【0009】次に、上記ＦＡＸ－ＯＣＲシステムの動作

3

を説明する。ここで、営業店２０から振込依頼書を送信する場合を説明する。先ず、営業店２０では、顧客が記入した振込依頼書を窓口にて行員が受け取る（図２中、ステップ①）。これにより、行員は、ファクシミリ２１より、受け取った振込依頼書を通信回線７０を介して地区センタ４０に送信する（ステップ②）。

【００１０】地区センタ４０のＦＣＵ４１は、営業店２０のファクシミリ２１から発信した帳票データ（圧縮されたイメージデータ）を受信し、ディスク装置４４に格納する（ステップ③）と共に、そのデータをＭＣＵ４２に送出する。ＭＣＵ４２は、ＦＣＵ４１から送信された帳票データを認識装置４６に送信し、認識装置４６は帳票データの文字認識を行う。そして、ＭＣＵ４２は、認識装置４６で認識された結果（文字データ＋該当エリアのイメージデータ）を、ディスク装置４７に格納し（ステップ④）、検証修正部４３に送信する（ステップ⑤）。

【００１１】検証修正部４３では、例えば、ワークステーション４３１をオペレータが操作して、イメージデータとこれを認識したコードデータとを比較照合し、認識装置４６が、イメージデータの正常認識を行ったか否かを判断する。そして、正常認識された場合はそのまま、一方、認識が誤っていた場合には修正コードデータの生成が行われる。そして、このような処理結果は、別途にワークステーション４３１～４３３に設けられた図示しない記憶部に一時格納される。ＭＣＵ４２は、正しく認識された、あるいは正しく修正されたコードデータを通信制御装置５０を介してホスト６０に転送する（ステップ⑥）。このようにしてホスト６０が、振込依頼書に記載された文字に対応する文字コードを受信すると、その後、営業店１０～３０からの依頼に基づく振替処理が実行される。

【００１２】各営業店１０～３０からの帳票データの送信は上記のように地区センタ４０を介して行われるが、地区センタ４０において、ＭＣＵ４２に障害が発生することがある。このような場合、営業店１０～３０からのデータ処理を行うことができないため、以下のようなリカバリ処理を行う。

【００１３】図３は、ＭＣＵ４２障害発生時のリカバリ処理を示すフローチャートである。先ず、ＭＣＵ４２に何らかの障害が発生した場合（ステップＳ１）、ＦＣＵ４１は、営業店１０～３０からのファクシミリデータ受信を不可の状態とする（ステップＳ２）。尚、ＭＣＵ４２の障害とは、そのハードウェア故障やソフトウェアによる原因等で運用が停止した状態になることをいう。また、上記ステップＳ２で営業店からの受信不可にした時点では、通常、ディスク装置４４内には、受信したまま一時格納され、ＭＣＵ４２に送信していないイメージデータが滞留データａとして存在している。

【００１４】次に、ＭＣＵ４２の復旧処理が行われると

4

（ステップＳ３）、そのリカバリ方法は、いずれかのワークステーション４３１～４３３より、システム再開のオペレーションがなされ（ステップＳ４）、これによって運用が再開される。ワークステーション４３１～４３３よりシステム再開指示を受けたＭＣＵ４２は、ＦＣＵ４１に対し、営業店１０～３０から受信したままＭＣＵ４２に対して未送信となっている滞留データのクリア指示を送出する（ステップＳ５）。

【００１５】尚、ＦＣＵ４１での滞留データをクリアする理由は、例えば障害が発生してホスト６０の受付終了直前に復旧し、この時点でＦＣＵ４１からＭＣＵ４２へ送信する滞留データが多い場合、これをＭＣＵ４２や検証修正部４３等でホスト６０の受付終了までに処理できない可能性があるためである。

【００１６】ＦＣＵ４１は、このＭＣＵ４２からのクリア指示に基づき滞留データをクリアし（ステップＳ６）、一方、ＭＣＵ４２は、ＦＣＵ４１に対して対象となる営業店への復旧情報の送信を行う（ステップＳ７）。これにより、ＦＣＵ４１は、ＭＣＵ４２から受信した復旧情報をファクシミリ送信するためにイメージデータ編集を行い（ステップＳ８）、これを対象となる営業店１０～３０に送信する（ステップＳ９）。その後、ＦＣＵ４１は、営業店１０～３０からのファクシミリデータ受信可能な状態となる（ステップＳ１０）。

【００１７】図中のｂは、上記ステップＳ９で送信される復旧情報の帳票例を示している。この帳票には、ステップＳ１にてＭＣＵ４２が障害となった時点でＭＣＵ４２受付済みの最終ＦＡＸ通番と欠番情報を出力する。例えば、図示の例では、最終受付ＦＡＸ通番は「２００」であり、ＦＡＸ通番「１９１」～「１９３」、「１９７」の振込依頼書のデータが欠けていることを示している。

【００１８】図４に、ＦＡＸ通番を記入する振込依頼書の一例を示す。図中、２０１で示す部分がＦＡＸ通番（ＦＡＸ番号）が記入される枠である。このＦＡＸ通番とは、各営業店１０～３０が帳票をファクシミリ１１～３１で送信する際に採番した通番であり、帳票毎に採番される。従って、復旧情報を受信した営業店１０～３０では、その欠番情報によって、どの帳票がＦＣＵ４１でクリアされたかを把握することができる。

【００１９】そして、復旧通知を受けた営業店１０～３０は、ホスト６０の受付終了時刻までに時間がない場合は、ステップＳ６でクリアされたデータを図示省略した営業店端末よりホスト６０に対して直接発信し、また、ホスト６０の受付終了時刻までに時間がある場合は、営業店１０～３０のファクシミリ１１～３１より帳票の再送を行い、地区センタ４０でデータ処理を行ってホスト６０に発信する。

【００２０】

【発明が解決しようとする課題】しかしながら、上記の

ような障害リカバリ処理では、以下のような問題点があった。

(1) ホスト６０の受付終了時刻までに時間が十分にある場合、ＦＣＵ滞留データを営業店１０～３０から再送するための処理時間および回線使用料金が無駄である。即ち、この場合は、一旦クリアしたデータと同一のデータをもう一度送信するといった無駄な処理を行わなくてはならなかった。

【００２１】(2) ＭＣＵ４２の復旧までに時間がかかる場合、営業店１０～３０に復旧通知が出力される時間が遅くなり、その結果、営業店１０～３０の対応が遅れ、ホスト６０への未送信のデータが発生してしまうといった重大な事故につながる恐れがあった。即ち、当日中に行うべき振込処理を行えないことは、銀行等の金融機関ではあってはならないことであり、このような障害への確実な対応策が求められていた。

【００２２】(3) 営業店１０～３０では、地区センタ４０の状況が分からないため、たとえＭＣＵ４２で障害が発生した場合でも、その状況に応じた対策を直ちにとることができない。即ち、地区センタ４０で障害が発生しても、営業店１０～３０では、帳票データが正常送信できないことが分かるだけで、その原因は判定できないため、地区センタ４０からの復旧通知を待つか、あるいは電話にて地区センタ４０に確認を行わない限り、早急な対応をとることができなかった。

本発明は、上記従来の問題点を解決するためになされたもので、効率がよくかつ信頼性の高い障害リカバリ制御システムを提供することを目的とする。

【００２３】

【課題を解決するための手段】第１発明の障害リカバリ制御システムは、複数のデータ送信元と、前記複数のデータ送信元と通信回線を介して接続され、各々のデータ送信元からのデータを蓄積し、データ処理を行う管理センタとからなるシステムにおいて、前記管理センタは、当該管理センタでの障害発生を想定し、その障害発生時刻に対応したリカバリ処理のパターンを予め設定したリカバリパターンテーブルと、前記管理センタで障害が発生した場合、前記リカバリパターンテーブルを参照し、障害発生時刻に対応したリカバリ処理を行うリカバリ処理部とを備えたことを特徴とするものである。

【００２４】第２発明の障害リカバリ制御システムは、上記第１発明において、リカバリパターンテーブルは、管理センタでの障害発生を想定し、その障害発生時刻と、データ送信元からのデータ送信先とに対応してリカバリ処理のパターンを予め設定したことを特徴とするものである。

【００２５】第３発明の障害リカバリ制御システムは、上記第１または第２発明において、リカバリ処理部は、障害発生後にデータ受信を再開するか否かを制御する受信制御と、障害発生までの未処理データをデータ送信元に返送するか否かを制御する返送制御とに基づくパターンで動作することを特徴とするものである。

【００２６】第４発明の障害リカバリ制御システムは、上記第１または第２発明において、リカバリ処理部は、障害発生後にデータ受信を再開するか否かを制御する受信制御と、障害発生までの未処理データをデータ送信元に返送するか否かを制御する返送制御とに基づきパターンを決定すると共に、複数のデータ送信元に対して各パターンに対応した同報メッセージを出力することを特徴とするものである。

【００２７】

【作用】第１発明の障害リカバリ制御システムにおいては、地区センタのＭＣＵで障害が発生した場合、ＦＣＵのリカバリ処理部は、先ず現在時刻をリードし、次にリカバリパターンテーブルからこの現在時刻に対応したリカバリ処理のパターンを求める。そして、リカバリ処理部は、求めたパターンのリカバリ処理を実行する。

【００２８】また、第２発明の障害リカバリ制御システムにおいては、データ送信元からのデータが、その送信先に対応して別々に格納されている。地区センタで障害が発生した場合、リカバリ処理部は、リカバリパターンテーブルを参照し、現在時刻と送信先とに基づき、リカバリ処理のパターンを求める。そして、リカバリ処理部は、送信先毎に求めたパターンでリカバリ処理を行う。

【００２９】更に、第３発明の障害リカバリ制御システムにおいては、障害が発生した場合、リカバリ処理部は、例えば、ホストの受付終了時刻までに十分時間のある場合は、一旦データ送信元からのデータ受信受付を停止し、その後障害が復旧すると、データ受信受付を再開する。また、この場合は障害発生までにデータ送信元から受信したデータはデータ送信元には返送しない。そして、障害発生時刻がホストの受付終了までにある程度余裕がある時刻であった場合、それ以降のデータ送信元からのデータ受信は停止し、地区センタでの未処理データは地区センタで処理する。また、障害発生時刻がホストの受付終了直前であった場合は、障害発生までに受信し、地区センタで処理していないデータは、データ送信元に返送され、データ送信元で処理される。

【００３０】第４発明の障害リカバリ制御システムにおいては、上記第３発明と同様のリカバリ処理を行うと共に、そのリカバリパターンに対応した同報メッセージを出力する。例えば、ホストの受付終了時刻までに十分時間のある場合は、暫くデータ送信を停止する旨のメッセージを送出し、ホストの受付終了時刻までにある程度時間がある場合は、未送信データはデータ送信元で処理する旨のメッセージとなり、また、ホスト受付終了直前であった場合は、地区センタでの未処理データを返送する旨のメッセージを送出する。

【００３１】

【実施例】以下、本発明の実施例を図面を用いて詳細に

説明する。図1は本発明の障害リカバリ制御システムを金融機関におけるFAX－OCRシステムに適用した場合の実施例を示すブロック図である。図において、営業店10～30にはデータ送信元としてのファクシミリ（FAX）11～31が設けられ、従来と同様に、これらファクシミリ11～31からのイメージデータが通信回線70を介して地区センタ40に送信されるようになっている。

【0032】また、ファクシミリ11～31からのデータを蓄積し、そのデータ処理を行う管理センタである地区センタ40は、ファクシミリ受信制御装置（FCU）410、制御部（MCU）42、検証修正部43、ディスク装置44、ワークステーション（WS）45、認識装置46、ディスク装置47、プリンタ48、ファクシミリ49で構成されている。ここで、FCU410には、本実施例の特徴点であるリカバリパターンテーブル101と、リカバリ処理部102が設けられている。リカバリパターンテーブル101は、地区センタ40のMCU42に障害が発生したと仮定した場合、その障害発生時刻に対応したリカバリ処理の種類を予め示すテーブルである。

【0033】図5は、このリカバリパターンテーブル101の内容とリカバリ処理実行のフローチャートを示す図である。即ち、リカバリパターンテーブル101は、障害発生時刻の時間帯が4分割され、これらの時間帯に対応したパターンが各々設定されている。例えば、時間帯が8：00～12：00ではパターンA、12：00～13：00ではパターンB、13：00～15：00ではパターンC、15：00～17：00ではパターンAといったように予めパターンが決定されている。尚、このパターンA～Cの内容については、後で詳細に説明する。

【0034】図1において、リカバリ処理部102は、MCU42で障害が発生した場合、リカバリパターンテーブル101を参照し、その障害発生時刻に対応したリカバリ処理を行う機能を有している。また、その他のFCU410の機能および地区センタ40における他の各構成は、従来の構成と同様であるため、ここでの説明は省略する。更に、通信制御装置（CCL）50、ホスト（HOST）60、通信回線70の構成も従来と同様である。

【0035】次に上記構成のFAX－OCRシステムにおける障害発生時の動作を説明する。尚、FAX－OCRシステムのデータ処理の流れ、即ち図中ステップ①～⑥で示す動作は、従来と同様であるため、ここでの説明は省略する。地区センタ40のMCU42で何らかの障害が発生した場合、FCU410は図5に示すようにリカバリ処理を行う。

【0036】先ず、FCU410がMCU42の障害を検出すると（ステップS1）、FCU410のリカバリ処理部102は、図示省略したFCU410内のクロックより現在時刻をリードする（ステップS2）。次いで、リカバリ処理部102は、リカバリパターンテーブル101をリードし（ステップS3）、ステップS2でリードした現在時刻からリカバリパターンを決定する（ステップS4）。そして、リカバリ処理部102は、決定したリカバリ処理を実行する（ステップS5）。

【0037】次に、上記リカバリパターンA～Cについて説明する。

《第1のパターン》図6は、第1のパターン（パターンA）を説明するためのフローチャートである。このパターンAは、ホスト60の受付終了までに十分時間のある場合である。尚、図5に示したリカバリパターンテーブル101において、時間帯15：00～17：00がパターンAとなっているのは、通常、他行宛の振込処理は15：00頃までで当日処理が打ち切られるため、即ち、他行間とのホストの受付が終了するため、これ以降の処理は同行間のみとなり、この場合は、時間的に十分余裕があるからである。

【0038】MCU42に障害が発生し（ステップS1）、FCU410がこれを検出すると（ステップS2）、リカバリ処理部102は、先ず、第1パターンの同報メッセージの文言をリードする（ステップS3）。尚、この同報メッセージの文言や後述する復旧メッセージ等の文言は、システム立上げ時にMCU42から転送されたり、予めFCU410内の図示しないメモリあるいはディスク装置44に格納されているものである。

【0039】ステップS2で第1パターンの文言をリードすると、リカバリ処理部102は、その同報メッセージを各営業店10～30宛に送信する（ステップS4）。各営業店10～30では、この同報メッセージが各ファクシミリ11～31で受信され、これが出力される（ステップS5）。図中のbがこの同報メッセージ例であり、地区センタ40に障害が発生し、障害が復旧するまでは帳票の送信を停止するようメッセージが伝達される。

【0040】その後、MCU42の障害が復旧すると（ステップS6）、MCU42は復旧通知をFCU410に対して出力する（ステップS7）。これにより、FCU410のリカバリ処理部102は、ディスク装置44内に滞留していた帳票のイメージデータ（図中、aで示す）をMCU42に送信する（ステップS8）。次に、リカバリ処理部102は、復旧通知のメッセージをファクシミリ送信用のイメージデータに編集し（ステップS9）、これを各営業店10～30に送信する（ステップS10）。

【0041】各営業店10～30では、これをファクシミリ11～31で受信し、メッセージとして出力する（ステップS11）。図中のcが、この復旧メッセージであり、帳票の送信を再開できることが示されている。

このように、パターンAでは、障害発生前に各営業店１０～３０から地区センタ４０に送信した帳票データは地区センタ４０で処理するため、営業店１０～３０から再送する必要がない。

【００４２】《第２のパターン》図７は、第２のパターン（パターンB）を説明するためのフローチャートである。このパターンBは、ホスト６０の受付終了までにある程度の時間がある場合に障害となった際に有効なリカバリ処理である。

【００４３】先ず、ＭＣＵ４２に障害が発生し（ステップＳ１）、これをＦＣＵ４１０が検出すると（ステップＳ２）、リカバリ処理部１０２は第２パターンの文言をリードし（ステップＳ３）、これを各営業店１０～３０に同報メッセージとして送信する（ステップＳ４）。各営業店１０～３０では、これをファクシミリ１１～３１で受信し、メッセージとして出力する（ステップＳ５）。この同報メッセージを図中bに示す。即ち、障害発生後、地区センタ４０ではデータ処理の受付は行わず、それ以降の帳票データ処理は営業店１０～３０で処理する旨のメッセージである。

【００４４】一方、ＦＣＵ４１０のリカバリ処理部１０２は、障害発生までに受信していた滞留データａを、ステップＳ３でリードした文言を付加して地区センタ４０のファクシミリ４９に出力する（ステップＳ６）。

【００４５】図８に、ファクシミリ４９で出力した帳票のイメージを示す。この帳票のイメージデータは、図４で示した振込依頼書の記載内容を示すもので、図中、Ａが付加した文言である。

【００４６】これにより、地区センタ４０では、ＭＣＵ４２が復旧すると（ステップＳ７）、その復旧通知をＦＣＵ４１０に対して送信すると共に（ステップＳ８）、ステップＳ６で出力された帳票のデータ処理を行う（ステップＳ９）。この帳票データ処理は、検証修正部４３のワークステーション４３１～４３３よりデータエントリ、即ちオペレータによる手入力による処理を行うか、あるいは地区センタ４０に設置されている別の端末装置でデータエントリを行う等の方法がある。以上のように、第２のパターン（パターンB）は、障害発生前に受信した滞留データを地区センタ４０でデータ処理することを特徴とするものである。

【００４７】《第３のパターン》図９は、第３のパターン（パターンC）を説明するためのフローチャートである。このパターンCは、ホスト６０の受付終了直前に障害となった場合に有効なリカバリ処理である。

【００４８】先ず、ＭＣＵ４２に障害が発生し（ステップＳ１）、これをＦＣＵ４１０が検出すると（ステップＳ２）、リカバリ処理部１０２は第３パターンの文言をリードし（ステップＳ３）、これを各営業店１０～３０に同報メッセージとして送信する（ステップＳ４）。また、障害発生前に受信した滞留データａを返送する（ステップＳ５）。各営業店１０～３０では、これらの送信データをファクシミリ１１～３１で受信して出力する（ステップＳ６、Ｓ７）。

【００４９】ステップＳ６で出力した同報メッセージを図中bに示す。即ち、営業店１０～３０から送信する予定であった未送信帳票と返送帳票は営業店１０～３０で処理するよう要請する旨のメッセージとなっている。また、返送帳票は以下の通りとなっている。

【００５０】図１０は、返送帳票例である。この帳票のイメージデータは、各ファクシミリ１１～３１から地区センタ４０に送信した振込依頼書の記載内容を示すもので、図中、Ａが付加されている文言である。これにより、各営業店１０～３０では、直接ホスト６０に対してデータを送信し、処理を終了する。以上のように、この第３のパターンは、障害発生以降にＭＣＵ４２にて未処理のデータは全て各営業店１０～３０で処理するものである。

【００５１】次に、リカバリパターンテーブル１０１を障害発生時刻だけでなく、データ送信先に対応させてリカバリ処理のパターンを設定した他の実施例を説明する。図１１は、他の実施例を説明するためのリカバリパターンテーブルとリカバリ処理のフローチャートである。即ち、この実施例では、他行宛の帳票送信と、自行宛の帳票送信とを電話番号（回線）を換えることによって地区センタ４０におけるＦＣＵ４１０がこれに応じたリカバリ処理を行うものである。例えば、ホスト６０の受付終了時刻の早い（通常は１５：００頃）他行宛の処理は８：００から１７：００までのパターンが「A、B、C、A」であり、ホスト６０の受付終了時刻の遅い自行宛の処理はそのパターンが「A、A、B、C」となっている。尚、このように回線７０ａと回線７０ｂとを用いることは、公衆網への申請で容易に対応が可能である。

【００５２】このように構成された他の実施例では、営業店１０～３０では、回線７０ａとして、電話番号（例えば、0484-31-1234）に他行宛の帳票を送信し（ステップＳ１）、回線７０ｂとして電話番号（例えば、0484-31-5678）に自行宛の帳票を送信する（ステップＳ２）。ＦＣＵ４１０では、これらの帳票データをディスク装置４４内の別のファイルにタンキング（打溜）する。

【００５３】そして、ＭＣＵ４２に障害が発生し、これをＦＣＵ４１０が検出すると（ステップＳ３）、リカバリ処理部１０２は現在時刻をリードし（ステップＳ４）、リカバリパターンテーブル１０１をリードする（ステップＳ５）。その後、リカバリ処理部１０２は、他行宛の滞留データａと自行宛の滞留データｂとに対応したリカバリパターンを決定する（ステップＳ６およびステップＳ７）。例えば、障害発生時刻が“１３：３０”であった場合、他行宛の滞留データａのリカバリパターンは「C」であり、自行宛の滞留データｂのリカバ

リパターンは「B」である。従って、他行宛の滞留データaは営業店10～30は返送され（ステップS8）、自行宛の滞留データbの場合は、MCU42の復旧待ちとなる（ステップS9）。即ち、自行宛の滞留データbの場合は、障害復旧後、FCU410よりMCU42に対して送信する。

【0054】尚、上記各実施例では、リガバリパターンテーブル101におけるリカバリパターンを3種類としたが、これに限定されるものではなく、更に多くのパターンとすることも可能であり、また、そのリカバリ処理についても、適用するシステムの形態に応じて種々変更してもよい。また、上記各実施例では、障害リカバリ制御システムを適用するシステムとして金融機関におけるFAX－OCRシステムの場合を説明したが、複数のデータ送信元からデータを管理センタに送信し、これらのデータを管理センタで処理するシステムであれば、上記各実施例と同様の効果を奏することができる。

【0055】

【発明の効果】以上説明したように、第1発明の障害リカバリ制御システムによれば、予め障害時刻に対応したリカバリ処理のパターンを設定し、障害が発生した場合は、このパターン別にリカバリ処理を行うようにしたので、データ再送等の必要がなく、効率が向上すると共に、障害に最適なリカバリ処理を行えるため、システムとしての信頼性も向上させることができる。

【0056】第2発明の障害リカバリ制御システムによれば、リカバリ処理のパターンをデータ送信先にも対応させて設定するようにしたので、第1発明の効果に加えて更に効率向上と信頼性向上を図ることができる。

【0057】第3発明の障害リカバリ制御システムによれば、リカバリ処理を、障害発生後管理センタ側で受信を再開するか、また、障害発生前の未処理データを送信元に返送するかといった制御のパターンとしたので、上記第1発明の効果に加えて、データ送信元でも障害発生後の処理を速やかにかつ確実に行うことができる。

【0058】第4発明の障害リカバリ制御システムによれば、障害発生時は、各データ送信元に対して各パターンに対応した同報メッセージを送出するようにしたので、データ送信元でも管理センタの状況が分かり、その後の処理への対応を速やかに行うことができる。

【図面の簡単な説明】

【図1】本発明の一実施例による障害リカバリ制御システムの適用例を示すブロック図である。

【図2】従来の障害リカバリ制御処理を説明するためのシステムブロック図である。

【図3】従来の障害リカバリ処理のフローチャートである。

【図4】帳票の一例を示す図である。

【図5】本発明の障害リカバリ制御システムの動作フローチャートとリカバリパターンテーブルを示す図である。

【図6】本発明の障害リカバリ制御システムにおけるリカバリ処理の第1パターンを示す説明図である。

【図7】本発明の障害リカバリ制御システムにおけるリカバリ処理の第2パターンを示す説明図である。

【図8】本発明の障害リカバリ制御システムにおけるリカバリ処理の第2パターンでの出力帳票イメージを示す説明図である。

【図9】本発明の障害リカバリ制御システムにおけるリカバリ処理の第3パターンを示す説明図である。

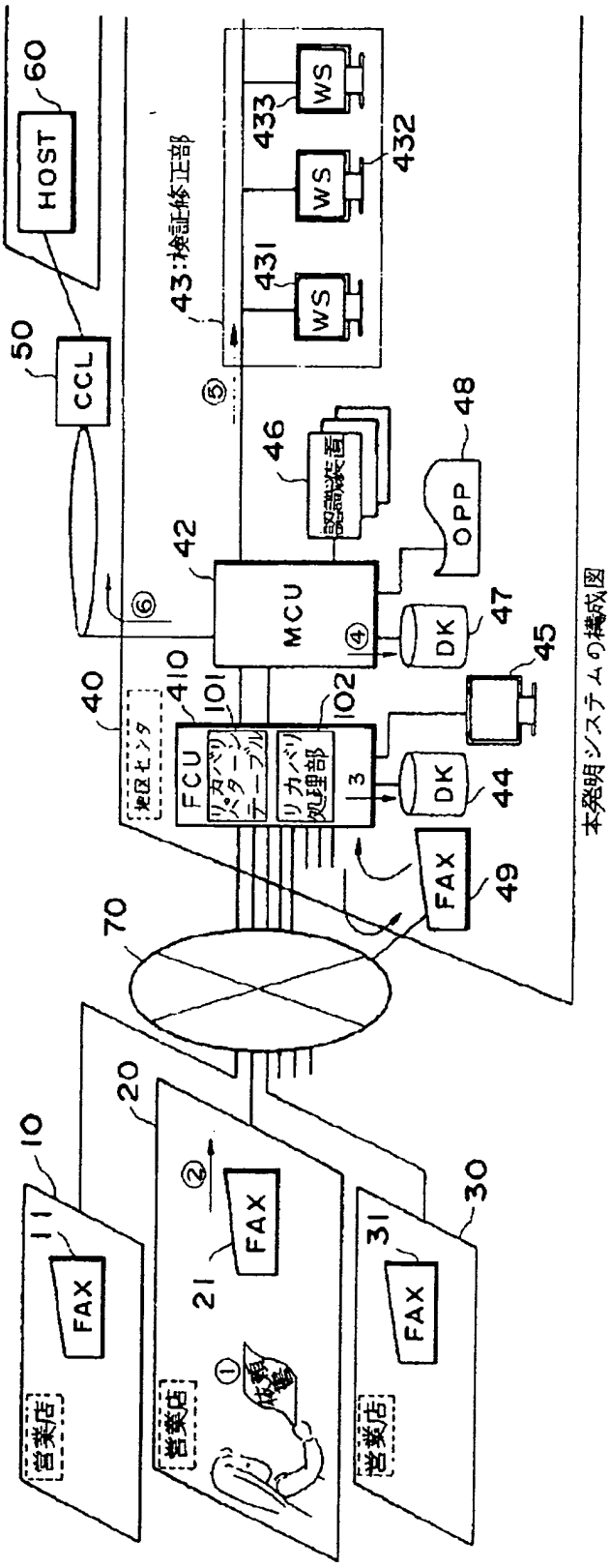
【図10】本発明の障害リカバリ制御システムにおけるリカバリ処理の第3パターンでの出力帳票イメージを示す説明図である。

【図11】本発明の障害リカバリ制御システムにおける他の実施例を示す説明図である。

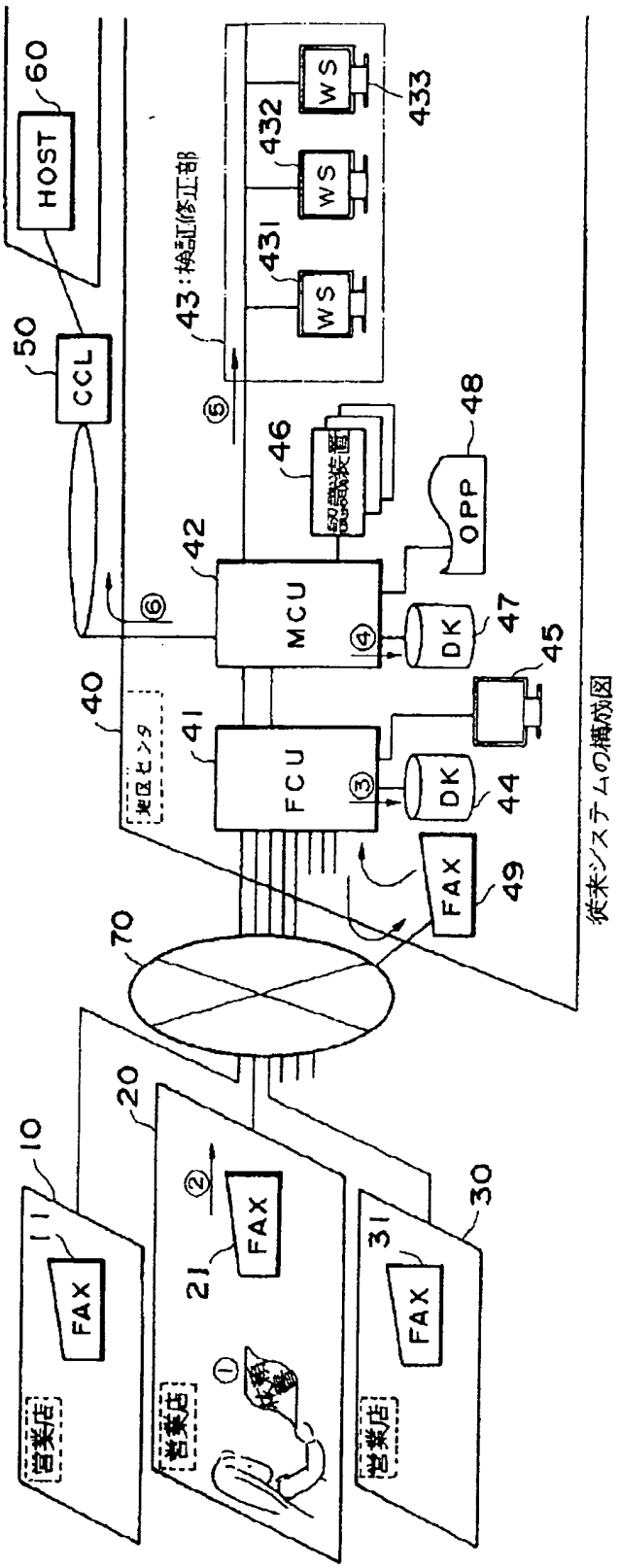
【符号の説明】

11、21、31　データ送信元（ファクシミリ）
40　管理センタ（地区センタ）
42　制御部
49　ファクシミリ
60　ホスト
101　リカバリパターンテーブル
102　リカバリ処理部

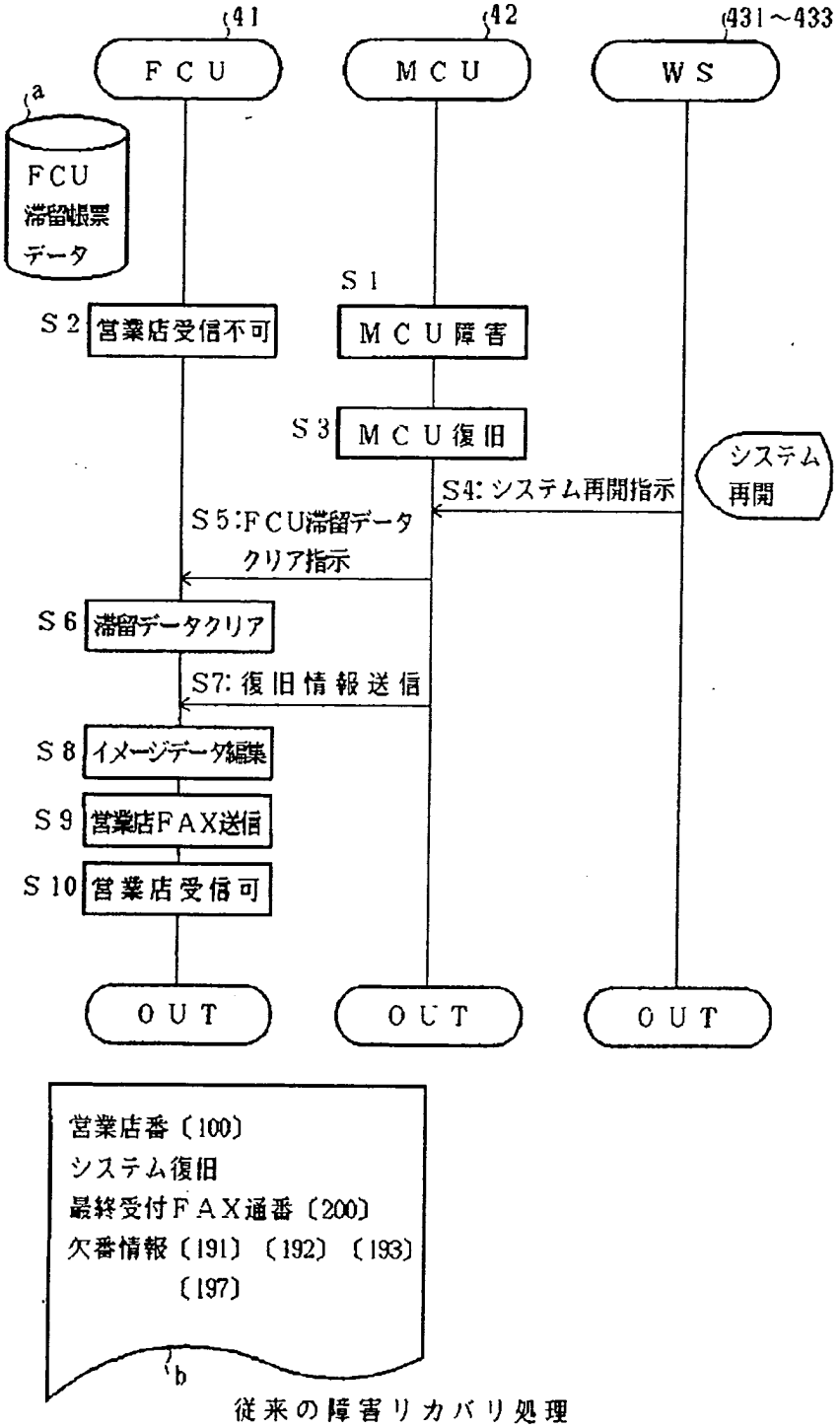
【図1】


HOST
60
CCL
50
43:検証修正部
WS
433
WS
432
WS
431
⑤
46
認識装置
48
OPP
42
MCU
④
DK
47
45
40
地区センタ
410
101
102
FCU
リカバリパターンテーブル
リカバリ処理部
13
DK
44
FAX
49
70
20
10
②
FAX
11
FAX
21
FAX
31
30
営業店
営業店
営業店
①
依頼書
⑥
本発明システムの構成図

【図2】

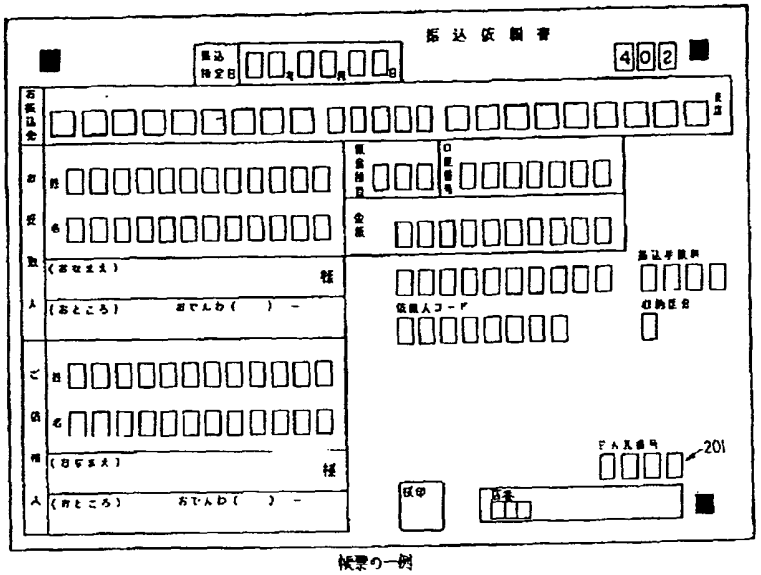

HOST
60
CCL
50
43：検証/修正部
431
432
433
WS
WS
WS
⑤
46
48
OPP
42
MCU
④
DK
47
45
40
拠点センタ
41
FCU
③
DK
44
FAX
49
70
従来システムの構成図
20
10
②
FAX
21
FAX
11
30
31
FAX
営業店
営業店
営業店
①

【図3】



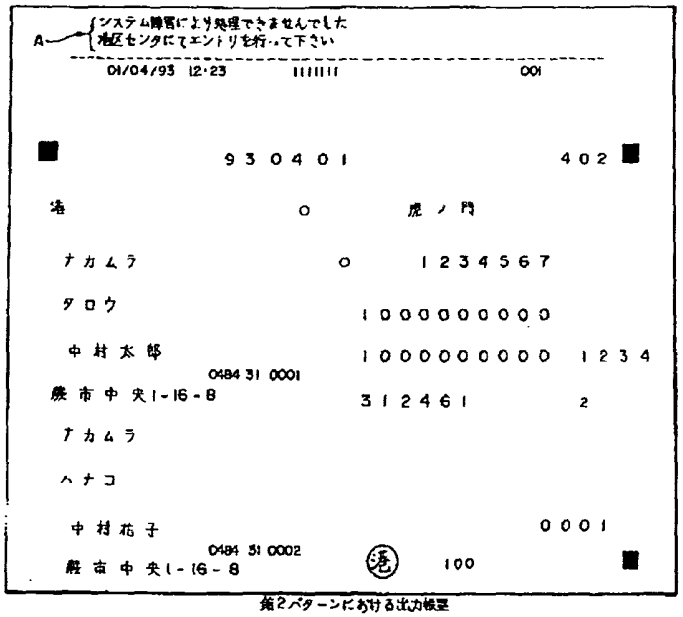
従来の障害リカバリ処理

【図4】



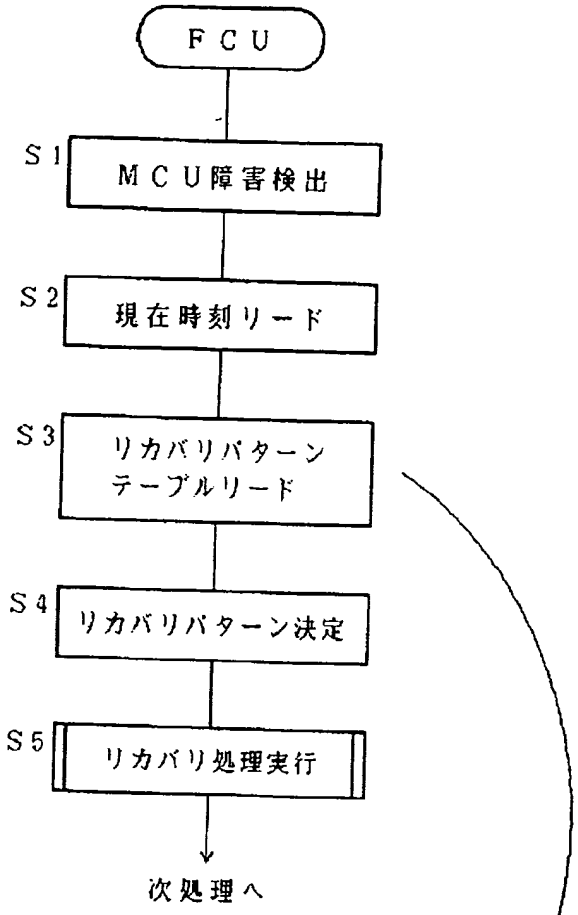
帳票の一例

【図8】



第2パターンにおける出力帳票

【図5】

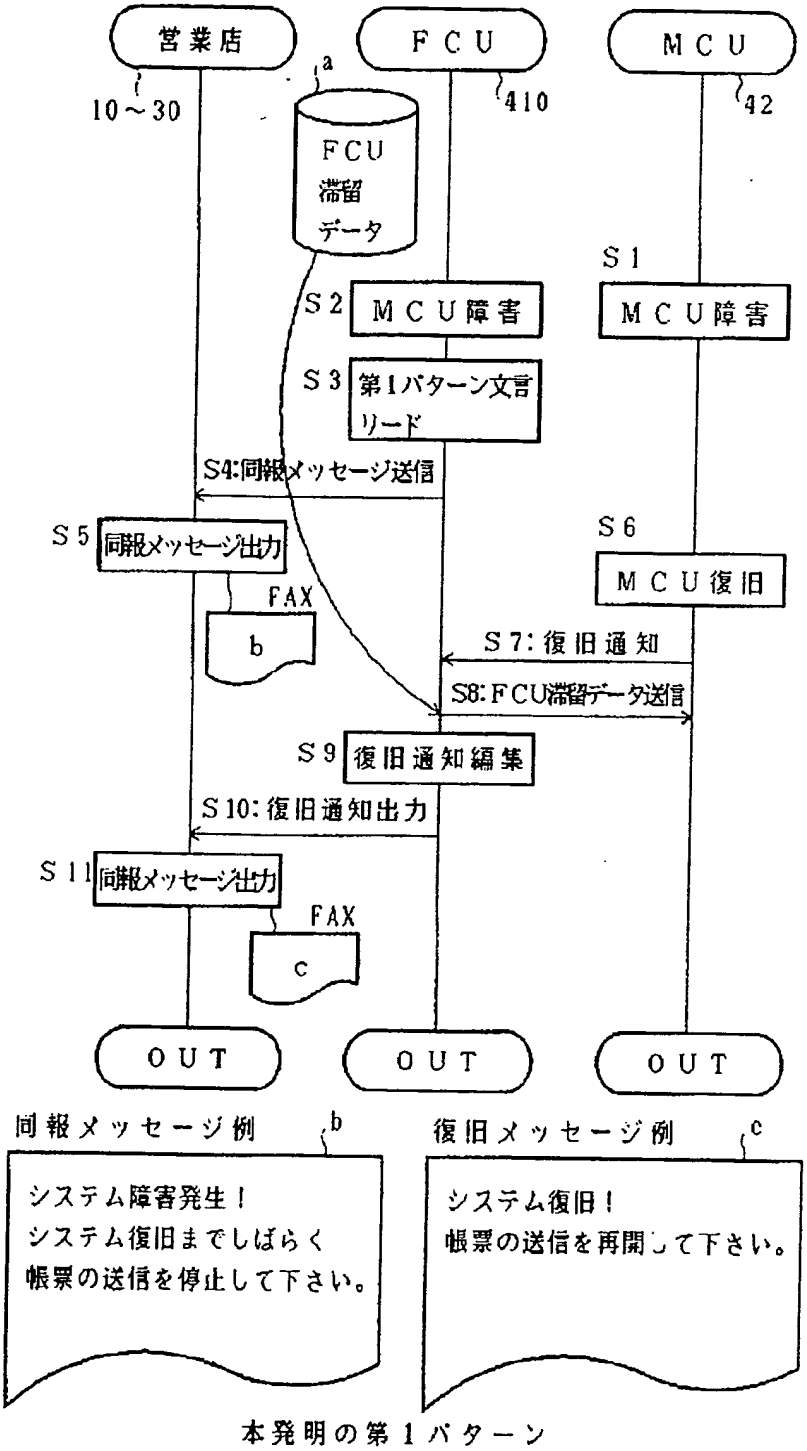


101:リカバリパターンテーブル

時間帯	パターン
8:00~12:00	A
12:00~13:00	B
13:00~15:00	C
15:00~17:00	A

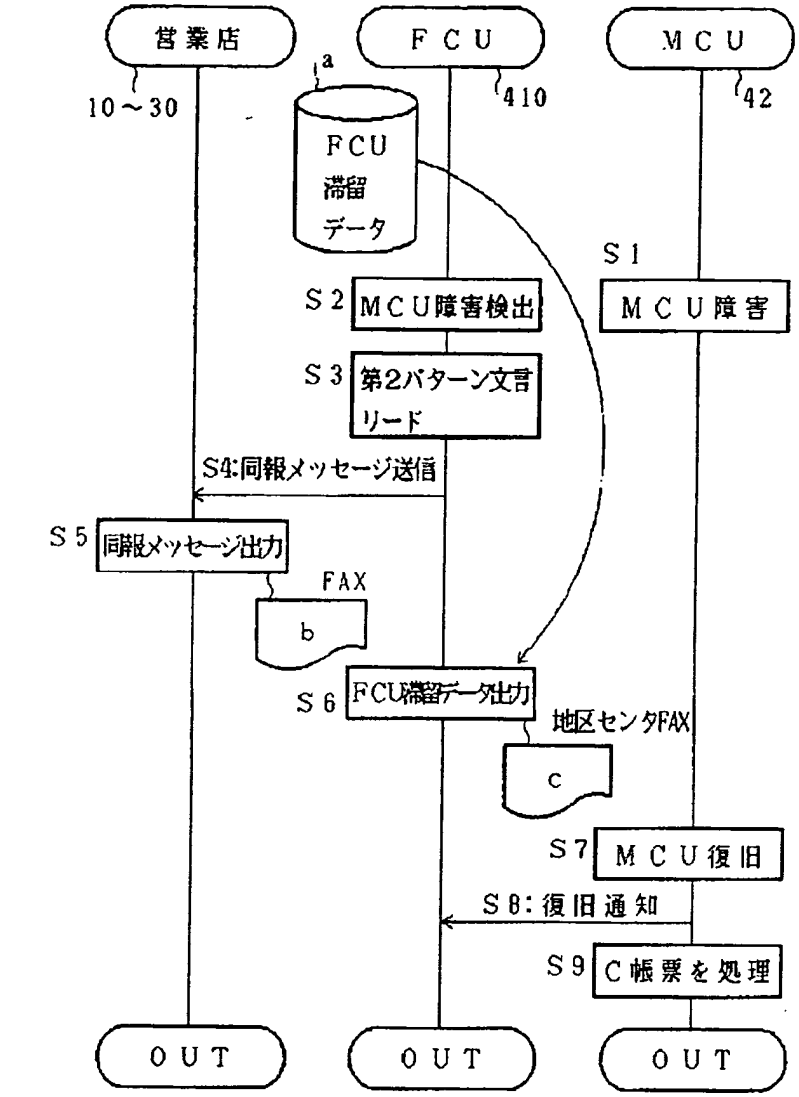
本発明のリカバリ処理とリカバリパターンテーブル

【図6】



本発明の第1パターン

【図7】

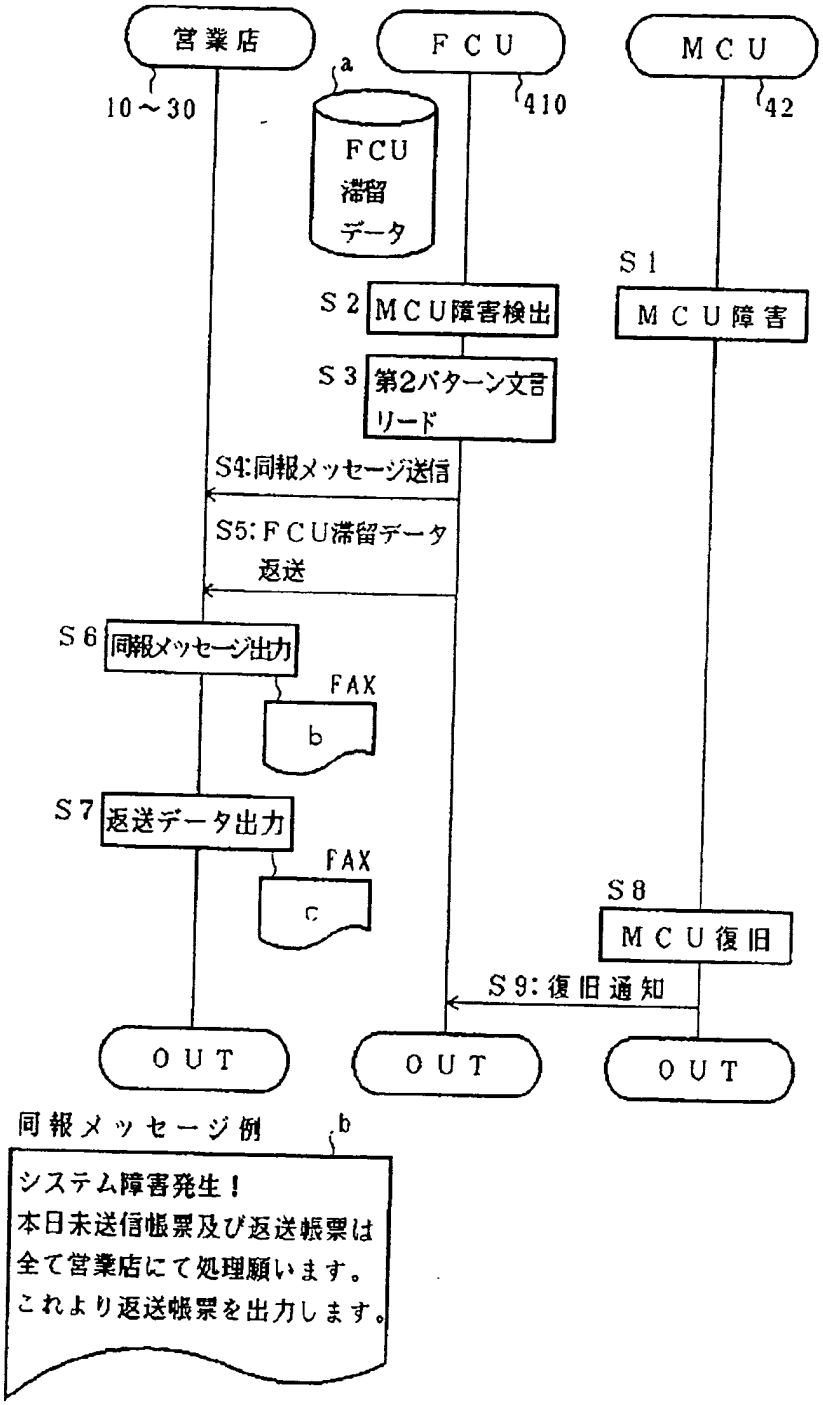


同報メッセージ例　b

システム障害発生！
本日未送信帳票は、営業店にて
処理願います。

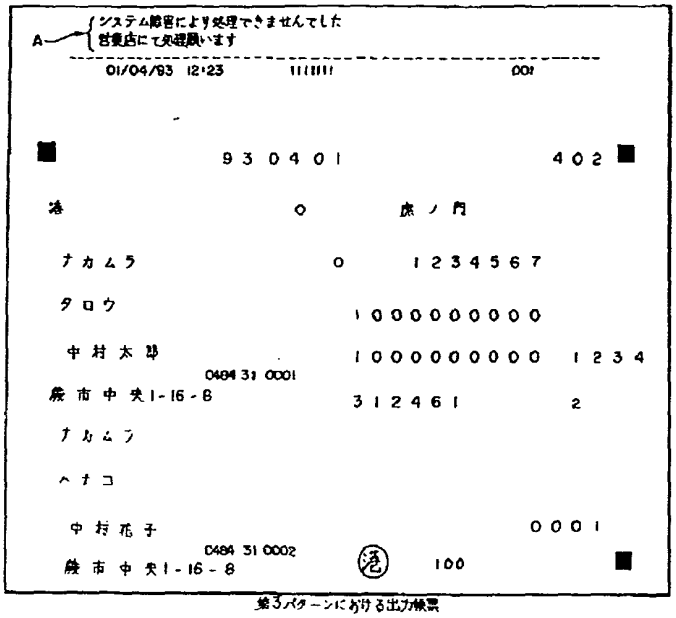
本発明の第２パターン

【図9】



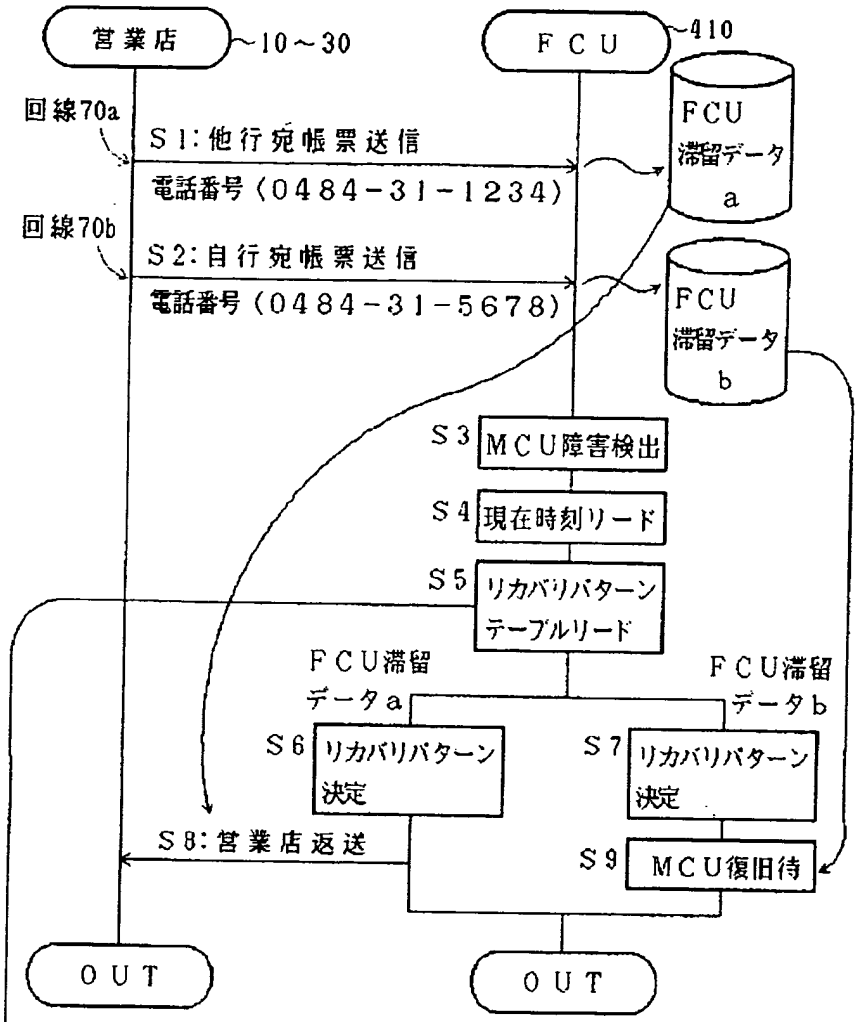
本発明の第３パターン

【図10】



第3パターンにおける出力帳票

【図11】



c:リカバリパターンテーブル

時間帯	パターン	
	回線70a	回線70b
8：00～12：00	A	A
12：00～13：00	B	A
13：00～15：00	C	B
15：00～17：00	A	C

他の実施例

Recovery Management in QuickSilver

ROGER HASKIN, YONI MALACHI, WAYNE SAWDON,
AND GREGORY CHAN
IBM Almaden Research Center

This paper describes QuickSilver, developed at the IBM Almaden Research Center, which uses *atomic transactions* as a unified failure recovery mechanism for a client-server structured distributed system. Transactions allow failure atomicity for related activities at a single server or at a number of independent servers. Rather than bundling transaction management into a dedicated language or recoverable object manager, QuickSilver exposes the basic commit protocol and log recovery primitives, allowing clients and servers to tailor their recovery techniques to their specific needs. Servers can implement their own log recovery protocols rather than being required to use a system-defined protocol. These decisions allow servers to make their own choices to balance simplicity, efficiency, and recoverability.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File System Management—*distributed file systems*; *file organization*; *maintenance*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; *checkpoint/restart*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Commit protocol, distributed systems, recovery, transactions

1. INTRODUCTION

The last several years have seen the emergence of two trends in operating system design: *extensibility*, the ability to support new functions and machine configurations without changes to the kernel; and *distribution*, partitioning computation and data across multiple computers. The QuickSilver distributed system, being developed at the IBM Almaden Research Center, is an example of such an extensible, distributed system. It is structured as a lean kernel above which system services are implemented as processes (*servers*) communicating with other requesting processes (*clients*) via a message-passing *interprocess communication* (*IPC*) mechanism. QuickSilver is intended to provide a computing environment for various people and projects in our laboratory, and to serve as a vehicle for research in operating systems and distributed processing.

One price of extensibility and distribution, as implemented in QuickSilver, is a more complicated set of failure modes, and the consequent necessity of dealing with them. Most services provided by traditional operating systems (e.g., file,

Authors' address: IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099.

display) are intrinsic pieces of the kernel. Process state is maintained in kernel tables, and the kernel contains cleanup code (e.g., to close files, reclaim memory, and get rid of process images after hardware or software failures). QuickSilver, however, is structured according to the client-server model, and as in many systems of its type, system services are implemented by user-level processes that maintain a substantial amount of client process state. Examples of this state are the open files, screen windows, and address space belonging to a process. Failure resilience in such an environment requires that clients and servers be aware of problems involving each other. Examples of the way one would like the system to behave include having files closed and windows removed from the screen when a client terminates, and having clients see bad return codes (rather than hanging) when a file server crashes. This motivates a number of design goals:

(1) Properly written programs (especially servers) should be resilient to external process and machine failures, and should be able to recover all resources associated with failed entities.
(2) Server processes should contain their own recovery code. The kernel should not make any distinction between system service processes and normal application processes.
(3) To avoid the proliferation of *ad-hoc* recovery mechanisms, there should be a uniform system-wide architecture for recovery management.
(4) A client may invoke several independent servers to perform a set of logically related activities (a *unit of work*) that must execute *atomically* in the presence of failures, that is, either all the related activities should occur or none of them should. The recovery mechanism should support this.

In QuickSilver, recovery is based on the database notion of *atomic transactions*, which are made available as a system service to be used by other, higher-level servers. This allows meeting all the above design goals. Using transaction-based recovery as a single, system-wide recovery paradigm created numerous design problems because of the widely different recovery demands of the various QuickSilver services. The solutions to these problems will be discussed in detail below. However, we will first discuss the general problem of recovery management and consider some alternative approaches.

1.1 Recovery from System and Process Failures

The problems with recovery in a system structured according to the client-server model arise from the fact that servers in general maintain *state* on behalf of clients, and failure resilience requires that each be aware of problems involving the other. Examples of this state are the open files, screen windows, and address space belonging to a client process. Examples of the way one would like the system to behave include having files closed and windows removed from the screen when a client terminates, and having clients see bad return codes (rather than hanging) when a file server crashes.

Timeouts. A simple approach to recovery is for clients to set timeouts on their requests to servers. One problem with this is that it substantially complicates the logic of the client program. Another obvious problem is the difficulty of

choosing the correct timeout value: excessively long timeouts impair performance and usability, whereas short timeouts cause false error signals. Both communication delays and server response time can be unpredictable. A database request may time out because of a crash, but the database server might also be heavily loaded, or the request (e.g., a large join) might just take a long time to execute. False timeouts can cause inconsistencies where the client thinks a request has failed and the server thinks it has succeeded.

Connectionless protocols. Several systems have attempted to define away the consistency problems of timeout-based recovery by requiring servers to be connectionless, stateless, and idempotent [9, 22]. A client that sees a timeout for an uncompleted request has the option of retrying or of giving up. Servers keep no state or only "soft" state, such as buffers that are eventually retired by an LRU policy. We think the stateless model to be unworkable for several reasons. Some state, such as locks on open files or the contents of windows, is inherently "hard." Some services, such as graphics output to intersecting areas, require requests to be sequenced and not to be repeated. Furthermore, the server's semantics may require several client requests to be processed atomically. The client giving up in the middle of a sequence of related requests can cause loss of consistency.

Virtual circuits. Consistency and atomicity problems are partially solved by employing connection-oriented protocols, such as LU6.2 sessions [17]. Failures are detected by the communications system, which returns an out-of-band signal to both ends. Atomicity and consistency can be achieved within a virtual circuit via protocols built on top of it. The primary limitation of virtual circuits is that they fail independently, thus multiserver atomicity cannot be directly achieved.

Some systems use hybrid recovery techniques that fall somewhere between timeouts and virtual circuits. In the V-System [9], recovery is done by detecting process failures. The kernel completes outstanding client requests to failed servers with a bad return code. Servers periodically execute **ValidPid** calls to determine the state of processes for which they are maintaining state. If the process has failed, the state is cleaned up. V has no system-defined atomic error recovery, although an architecture for implementing this at the client level via runtime library functions has been proposed [10].

Replication. Another approach to failure resilience is through replication. Clients are presented with the view of a reliable and available underlying system. Examples of systems that use replication are Locus [29], which replicates the file system; ISIS [5] and Eden [30], which replicate storage objects; and Circus [12] and Tandem [4], which replicate processes. Replication simplifies the life of clients, and eliminates the need for them to detect and recover from server failures. However, replication is too expensive to use for some system services, and does not make sense in others (e.g., display management). Furthermore, to implement replication, servers still have to be able to detect failures and coordinate their recovery. Thus, replicated systems are usually built on top of a transaction mechanism. Given our desire to have a single recovery mechanism

institutionalized in the system, we thought transactions to be the better choice of the two.

1.2 Transactions

Previous work. There is a substantial body of literature relating to transaction-based recovery in the context of single services, such as file systems [38] and databases [15]. The applicability of the concept has been explored in the context of both local [15] and distributed [19, 32] systems. More recently, there have been several experiments with using transactions as a general recovery mechanism for an operating system. Argus [21], for example, provides language constructs for recoverable shared objects, and provides underlying system facilities for implementing these constructs. TABS [34] provides transaction management as a service running under Accent [31], and allows it to be used by *data servers* to enable them to implement recoverable objects callable by Accent messages. More recently, Camelot [36] provides a similar level of function running on Mach [3]. We will defer comparing QuickSilver to these systems until later in the paper.

Recovery demands of various servers. A painful fact is that transactions, as they are normally thought of, are a rather heavyweight mechanism. Using transactions as a single, system-wide recovery paradigm depends upon being able to accommodate simple servers in an efficient way. To get a feel for this, let us examine the characteristics of a few representative servers in QuickSilver.

The simplest class of servers are those that have *volatile* internal state, such as the window manager, virtual terminal service, and address space manager (loader). For example, the contents of windows does not survive system crashes. These servers only require a signalling mechanism to inform them of client termination and failures. Often, such servers have stringent performance demands. If telling the loader to clean up an address space is expensive, command scripts will execute slowly.

A more complex class of servers manages *replicated, volatile state.* An example is the name server that other QuickSilver servers use to register their IPC addresses. To maximize availability, this server is replicated, and updates are applied atomically to all replicas. The state of each replica is volatile (i.e., not backed up in stable storage). This is conceptually similar to Synchronous Global Memory (a. k. a. delta-common storage) [13] and *troupes* [12]. Individual replicas recover by querying the internal state of a functioning replica. The exceedingly rare catastrophic simultaneous failure of all replicas is recovered from by having servers re-register themselves. Replicated volatile state uses transaction commit to provide atomicity, yielding a useful increase in availability without the expense of replicated stable storage.

The services that require the most from the recovery manager are those that manage *recoverable state*, such as QuickSilver's transaction-based distributed file system [7]. The file system uses transactions to recover from server crashes, and to detect and recover from client crashes. Furthermore, since the file system is structured as a federation of independent servers on different nodes, the transaction mechanism provides atomicity across these servers. Finally, commit

coordination and recovery can be provided atomically between the file system and other servers (e.g., database) that might exist in the future.

A final class of users are not servers at all. Long-running application programs with large data sections (e.g., simulations), whose running time may exceed the mean time between failures of a machine, require a checkpoint facility to make their state recoverable. Just as logging can be superior to shadowing in a database system [15], incrementally logging checkpoint data may be superior to dumping the entire data section to a file. Checkpointable applications use the log directly, without using commit coordination.

1.3 A Transaction-Based Recovery Manager

The QuickSilver recovery manager is implemented as a server process, and contains three primary components:

(1) **Transaction Manager.** A component that manages commit coordination by communicating with servers at its own node and with transaction managers at other nodes.

(2) **Log Manager.** A component that serves as a common recovery log both for the Transaction Manager's commit log and server's recovery data.

(3) **Deadlock Detector.** A component that detects global deadlocks and resolves them by aborting offending transactions.

Of these three components, the Transaction Manager and Log Manager have been implemented and are in use, and will be discussed in detail. The Deadlock Detector, based on a design described by Obermarck [27], has not been implemented, but is mentioned here to show where it fits into our architecture.

The basic idea behind recovery management in QuickSilver is as follows: clients and servers interact using IPC messages. Every IPC message belongs to a uniquely identified transaction, and is tagged with its transaction ID (*Tid*). Servers tag the state they maintain on behalf of a transaction with its *Tid*. IPC keeps track of all servers receiving messages belonging to a transaction, so that the Transaction Manager (TM) can include them in the commit protocol. TM's commit protocol is driven by calls from the client and servers, and by failure notifications from the kernel. Servers use the commit protocol messages as a signalling mechanism to inform them of failures, and as a synchronization mechanism for achieving atomicity. Recoverable servers call the Log Manager (LM) to store their recovery data and to recover their state after crashes.

The recovery manager has several important properties that help it address its conflicting goals of generality and efficiency. Although the remainder of this paper describes these properties in detail, they bear mentioning now:

—The recovery manager concentrates recovery functions in one place, eliminating duplicated or ad hoc recovery code in each server.

—Recovery management primitives (commit coordination, log recovery, deadlock detection) are made available directly, and servers can use them independently according to their needs.

—The transaction manager allows servers to select among several variants of the commit protocol (one-phase, two-phase). Simple servers can use a

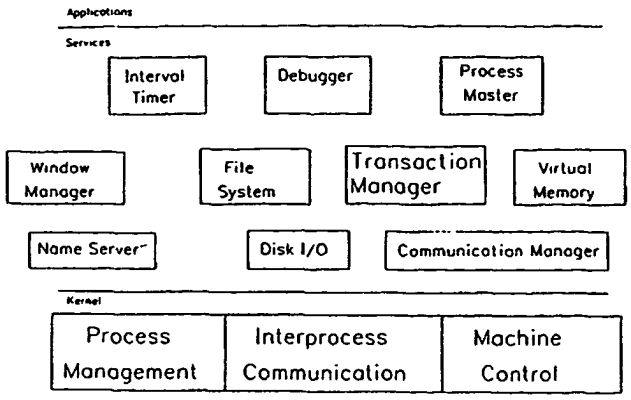


Fig. 1. QuickSilver system structure.

lightweight variant of the protocol, while recoverable servers can use full two-phase commit.

—Servers communicate with the recovery manager at their node. Recovery managers communicate among themselves over the network to perform distributed commit. This reduces the number of commit protocol network messages. Furthermore, the distributed commit protocol is optimized (e.g., when all servers at a node are one-phase or read only) to minimize log forces and network messages.

—The commit protocols support mutual dependencies among groups of servers involved in a transaction, and allows asynchronous operation of the servers.

—The log manager maintains a common log, and records are written sequentially. Synchronous log I/O is minimized, because servers can depend on TM's commit record to force their log records.

—A block-level log interface is provided for servers that generate large amounts of log traffic, minimizing the overhead of writing log records.

—Log recovery is driven by the server, not by LM. This allows servers to implement whatever recovery policy they want, and simplifies porting servers with existing log recovery techniques to the system.

2. QUICKSILVER ARCHITECTURE

A detailed discussion of the QuickSilver recovery management architecture requires some familiarity with the system architecture. Figure 1 shows the basic structure of QuickSilver. All services are implemented as processes and, with a few exceptions,[1] are loaded from the file system. Services perform both high-level functions, such as managing files and windows, and low-level device driver functions. The kernel contains three basic components: process management (creation, destruction, and dispatching), machine control (initialization, invoking interrupt handlers in user processes), and interprocess communication (IPC).

[1] The exceptions are the services used to create address spaces, processes, and to load programs, namely Process Master (loader), Virtual Memory, Disk I/O, and the File System itself.

Applications use IPC to invoke services, and services use IPC to communicate with each other.[2] Shared memory is supported among processes implementing multithreaded applications or servers, but is not used between applications and services or across any domain that might be distributed.

2.1 Interprocess Communication

QuickSilver IPC is a request-response protocol structured according to the client-server model. The basic notion is the *service*, which is a queue managed by the kernel of the node on which the service is created. Each service has a globally unique *service address* that can be used to send *requests* to the service. A service can be used for private communication between sets of processes, or can be made publicly accessible by registering it with the *name server*, which has a well-known service address. A process that wishes to handle requests to a service (a *server*), *offers* the service, establishing a binding between the service and a piece of code (the *service routine*) inside the server. When the server enters an inactive state by calling *wait*, the kernel attempts to match incoming requests to the offer, at which point the service routine will be invoked. The server can either *complete* the request (which sends the results to the client) in the service routine, or can queue the request internally and complete it later. Server processes must execute on the node at which the service was created. More than one process can offer a service, but since there is no method of directing successive requests from the same client to the same server process, the servers must cooperate to handle such requests (e.g., via a shared address space).

Client processes can issue any of three kinds of requests: *synchronous*, *asynchronous*, or *message*. Synchronous requests block the client until the server completes the request. Asynchronous requests are nonblocking and return a *request ID* that the client can use later to wait for completion. Message requests are also nonblocking, but cannot be waited on. QuickSilver IPC supports *multiple wait*: requests and/or offers can be combined into *groups*. Waiting on a group suspends the process until either a request is completed or an offer is matched to an incoming request.

QuickSilver makes several guarantees regarding the reliability of IPC: requests are not lost or duplicated, data is transferred reliably, and a particular client's requests are queued to the service in the sequence they are issued. Requests are matched to waiting offers in the order they are queued, though as mentioned they are not necessarily completed in order. If a server process terminates before completing a request, the kernel completes it with a bad return code. IPC's semantics are location-transparent in that they are guaranteed regardless of whether the request is issued to a local or remote service.

2.1.1 *Remote IPC.* When a client and server are on the same node, the kernel handles matching requests to offers, dispatching the server, and moving parameter data. When a request is made to a server on a remote node (determined by examining the service address), the kernel forwards the request to the *Communication Manager* (*CM*), a server that manages remote IPC communications (see

[2] Normally, programs issue requests by calling runtime-library stubs, which hide the details of parameter marshalling and message construction from the caller.

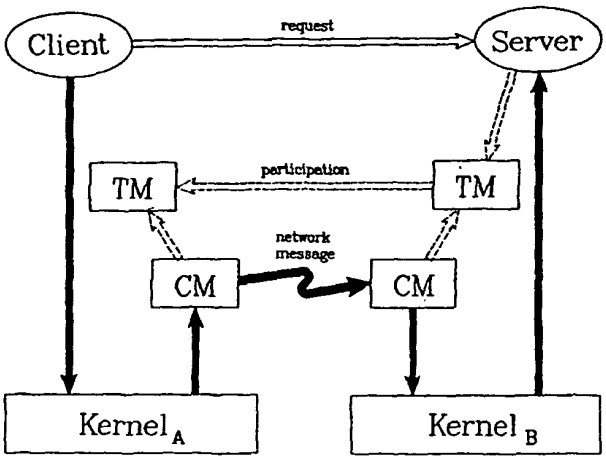


Fig. 2. QuickSilver distributed IPC.

Figure 2). CM implements the location transparent properties of IPC by managing routing, error recovery, and flow control. All IPC traffic between a pair of nodes is multiplexed over one connection maintained by the CMs on the two nodes. The CMs implement a reliable communication protocol that allows them to recover from intermittent errors (e.g., lost packets) and detect permanent ones (e.g., node or link failure), which are reported to TM. When CM detects a permanent failure of a connection to a node,[3] it causes all uncompleted requests to servers at that node to be completed with bad return codes.

3. TRANSACTION MANAGEMENT

This section describes how transactions work in QuickSilver, and the roles of clients, servers, and TMs. The commit coordination protocols are described in the next section. In QuickSilver, TM supports multisite atomicity and commit coordination. It does not manage serialization; this remains the responsibility of the servers that manage access to serializable resources. TM's knowledge of transactions is limited to those functions necessary for recovery.

Transactions are identified by a globally unique *Transaction Identifier* (*Tid*) consisting of two parts: the unique node ID of the transaction birth-site, and a sequence number that TM guarantees to be unique in each machine over time. Each IPC request in QuickSilver is made on behalf of a transaction and is tagged with its *Tid*. Run-time IPC stubs automatically tag requests they generate with a *Tid*, which defaults to one automatically created for the process when it begins, but which can be changed by the process. This allows simple clients to remain unaware of the transaction mechanism if they so choose. It is required (and enforced) that a process making a request on behalf of a transaction either be an *owner* of that transaction, or a *participant* in the transaction (both defined below). Servers tag all resources (state) they maintain on behalf of a transaction with the *Tid*. The mapping between transactions and resources enables the server to recover its state and reclaim resources when the transaction terminates.

[3] This implies either node or link failure.

Clients and servers access TM by calling run-time library stubs. These build the IPC requests to TM and return results after TM completes the request. Asynchronous variants of the stubs allow the caller to explicitly wait for the result. In the discussion below, "call" will be used to mean "send a request to."

3.1 Transaction Creation and Ownership

A process calls **Begin** to start a new transaction. TM creates the transaction, assigns it a *Tid*, and becomes the *coordinator* for the transaction. The caller becomes the transaction's *owner*. Ownership conveys the right to issue requests on behalf of the transaction and to call **Commit** or **Abort**. Clients can, if they wish, own and issue requests on behalf of any number of transactions.

The **ChangeOwner** call transfers ownership of a transaction to a different process. For example, the Process Master creates a new process, creates its default transaction, transfers ownership to the new process, and finally starts the process. Ownership spans the interval between the **Begin** or **ChangeOwner** call and the **Commit** or **Abort** call.

3.2 Participation in Transactions

When a server offers its service, it declares whether it is *stateless*, *volatile*, or *recoverable*. When a volatile or recoverable server receives a request made on behalf of a transaction it has not seen before, IPC registers the server as a *participant* in the transaction. Participants are included in the commit protocol, and have the right to themselves issue requests on behalf of the transaction. Participation spans the interval between receiving a request made on behalf of the transaction and responding to a **vote** request (see Section 4).

3.3 Distributed Transactions

A transaction becomes *distributed* when a request tagged with its *Tid* is issued to a remote node. When a process (client or server) at node A issues a request to a server at node B (S_B), IPC registers TM at node B (TM_B) as a *subordinate* of TM_A, and (as above) registers S_B as a participant with TM_B. Thus, the TM at each node coordinates the local activities of a transaction and the activities of subordinate TMs, and cooperates with superior TMs. The topology of a transaction can be thought of as a directed graph, with the coordinator at the root, TMs at the internal nodes, the owner and servers at the leaves, and arcs pointing in the direction of the superior-subordinate relation (see Figure 3).[4] There is no global knowledge of the transaction topology; each TM only knows its immediate superiors and subordinates. IPC assures that the graph is built with the invariant property that there is always a path connecting the coordinator to each node in the graph. This property is used to assure proper sequencing of operations during commit processing.

Using the graph topology rather than a single centralized coordinator was done for efficiency. The number of network messages is reduced both on the local

[4] This organization is similar to the hierarchy described in [23], with the exception that a TM can have multiple superiors and the graph can have cycles.

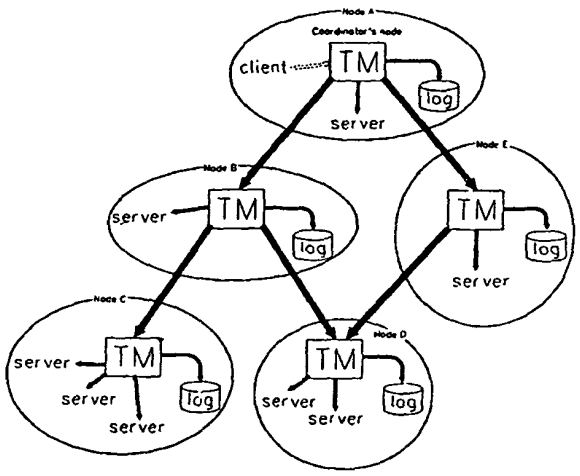


Fig. 3. Structure of a distributed transaction.

network (since servers communicate only with their node's TM using local IPC) and on the internet. For example, QuickSilver's distributed file system is structured such that a file's directory entry and data may reside on different nodes on the same local-area net. When a client accesses a file over the internet, the coordinator only communicates with one of the TMs (e.g., the directory manager's) over the internet; that TM then communicates with the other (e.g., the data manager's) over the LAN. This requires fewer internet messages than would be the case if all TMs communicated directly with the coordinator.

3.4 Transaction Termination and Failure

A TM *terminates* a transaction in response to one of the following conditions:

(1) The transaction's owner calls **Commit** or **Abort**.
(2) The owner process terminates. Normal termination is equivalent to **Commit**, and abnormal termination is equivalent to **Abort**.
(3) A participant calls **Abort**.
(4) A volatile or recoverable participant fails (i.e., terminates before voting).
(5) The local CM detects a permanent connection failure to a node whose TM is a superior in the transaction.
(6) A subordinate TM reports the termination of the transaction.

Any of these conditions cause the TM to initiate its commit processing.

A transaction can *fail* before it terminates. A failed transaction is not immediately terminated; instead, the failure is remembered and the transaction is aborted when it does terminate. This allows nonrecoverable operations (e.g., error reporting) to continue, but ensures that any further recoverable operations will be undone. A TM causes a transaction to fail under any of the following conditions:

(1) A volatile or recoverable participant fails (i.e., terminates before voting).

(2) The local CM detects a permanent connection failure to a node whose TM is a subordinate in the transaction.
(3) A subordinate TM reports the failure of the transaction.

Note that participant failure can cause either transaction failure or termination. Servers declare this when they offer their service. The asymmetry in failure of superior vs. subordinate nodes allows early reclamation of resources for the subordinate, while allowing the error to be seen and reported by the superior.

3.5 Transaction Checkpoints

The **Checkpoint** call allows the owner to save the partial results of a transaction. All the changes to state before the transaction checkpoint take effect permanently and, if the transaction later aborts, it will roll back only to the checkpoint. Servers retain any locks that have been acquired by the transaction, so consistency is maintained across checkpoints. Transaction checkpoints provide a means for long-running applications to survive system crashes and for distributed programs to synchronize themselves without the overhead of starting a new transaction at every synchronization point.

4. COMMIT PROCESSING

QuickSilver commit processing closely follows the distributed commit paradigm described in [23]. However, we will give a brief overview here to establish some terminology that will be used later. A TM initiates commit processing in response to a transaction termination condition.[5] To abort a transaction, a TM sends **abort** requests to all of its participants and immediate subordinate TMs; the latter recursively propagate the **abort** request to their participants and subordinates. In attempting to commit a transaction, the coordinator sends **vote** requests to all of its participants and immediate subordinate TMs; again, the latter recursively propagate the **vote** request. When a recipient of the **vote** request is *prepared* (recoverably ready to commit or abort), it returns its vote (**vote-commit** or **vote-abort**) by completing the request. To become prepared, a TM must receive **vote-commit** responses to each of its **vote** requests. When the coordinator is prepared, it commits the transaction and sends out **end** requests, which contain the outcome of the transaction (**end-commit**, **end-abort**) and get propagated in a manner similar to the **vote** requests. When all **end** requests have been completed, signifying that all involved parties know that the transaction has committed, the coordinator ends the transaction.

If a participant fails while prepared, it contacts its local TM after restarting to find out the status of the transaction. If a TM fails while prepared, it contacts its superior or the coordinator, whose identity is sent out with the **vote** requests and logged in the **prepare** record.

4.1 Basic Commit Protocols

QuickSilver supports two basic models for committing a transaction; **one-phase** and **two-phase** commit. Servers declare which protocol they follow when they offer their service.

[5] Note that only the coordinator can initiate a commit, but any subordinate TM can initiate an abort.

The one-phase protocol is used by servers that maintain only volatile state. TM sends an **end** request to each one-phase participant to notify it that the transaction is terminating. QuickSilver supports three variants of the one-phase protocol, allowing the server to be notified at different points in the commit processing. These are:

(1) **One-phase immediate.** The server is notified during the first (vote) phase of the two-phase protocol. An example is the window manager, which can reclaim its resources (windows) without waiting for commit processing to complete.
(2) **One-phase standard.** The server is notified when the transaction commits or aborts. Most servers use this variant.
(3) **One-phase delayed.** The server is notified when the commit processing has ended. An example server is CM, which cannot clean up its state (e.g., virtual circuits) until after the commit protocol has completed.

The two-phase protocol provides both synchronization and recoverability. The protocol used by QuickSilver is derived from the two-phase presumed-abort protocol described in [23]. Presumed abort has advantages that are important to QuickSilver servers, including its reduced cost for read-only transactions and the ability to forget a transaction after it ends. QuickSilver extends this protocol to distinguish between the synchronization and the recoverability it provides, and to accommodate the directed graph transaction topology. These extensions are discussed below.

4.2 Voting

QuickSilver defines four votes that a participant may return in response to a **vote** request:

(1) **Vote-abort.** The participant forces the transaction to be aborted. It may immediately undo its own actions and is not included in phase two of the commit protocol. The second phase is used to announce the abort to all other participants.
(2) **Vote-commit-read-only.** The participant votes to commit the transaction, declares that it has not modified any recoverable resources, and requests not to be included in phase two of the commit protocol.
(3) **Vote-commit-volatile.** The participant votes to commit the transaction, declares that it has not modified any recoverable resources, but requests to be informed of the outcome of the transaction during phase two.
(4) **Vote-commit-recoverable.** The participant votes to commit the transaction, declares that it *has* modified its recoverable state, and thus requests to be informed of the outcome of the transaction during phase two.

Vote-commit-volatile is an extension of the standard presumed-abort protocol of [23] that allows TM to provide less expensive synchronization for non-recoverable servers, such as those maintaining replicated volatile state, by

minimizing log activity. If no participants or subordinates respond **vote-commit-recoverable**, TM does not write any commit protocol log records.

4.3 Advanced Commit Protocols

QuickSilver guarantees atomicity will be preserved in case of process or machine failure, and even in the case of an improperly functioning client. In part, this is achieved by IPC, which guarantees delivery and ordering of requests, enforces that requests are issued on behalf of valid transactions by valid owners and/or participants, and keeps track of server participation to ensure that the transaction graph is properly connected. However, IPC does not guarantee ordering of requests outside of a single client-server conversation. Since transactions may involve several separate conversations between clients, servers, and TMs, it is still possible for the graph not to be fully formed and stable during commit processing. It is necessary that the commit protocol take this into account. This section discusses some of these problems and their solutions.

4.3.1 *Commit before Participate.* Consider the case where a client commits a transaction before all IPC requests made on its behalf are completed.[6] For example, suppose a client on node A calls a local server (S_{A_1}) and commits without waiting for S_{A_1} to complete the request. Furthermore, suppose S_{A_1} calls S_{A_2} as part of processing the client request. TM may see S_{A_2}'s participation after it has committed the transaction. In such a case, TM would tell S_{A_2} to abort (cf., due to presumed abort) even though the transaction committed. The simple expedient of forbidding the client to call **Commit** with uncompleted requests is not acceptable, since this is a normal state of affairs (e.g., requests for user input).

To ensure that all servers involved in a transaction participate in the commit protocol, TM, the kernel, and servers obey the following rules:

Rule 1. TM must accept new participants and include them in the voting until it commits.

Rule 2. Requests are partitioned into those that must complete before the transaction commits, and those (called "ω-requests") that need not complete because they do nothing that could force the transaction to abort.[7] TM at a node will not decide to commit a transaction until all non-ω-requests issued on the transaction's behalf on that node have completed.

Rule 3. A one-phase server that makes a non-ω-request on behalf of a client transaction (e.g., as part of servicing a request made to it) must make that request before completing the request it is servicing.

These rules are sufficient to ensure that TM will properly include all participants in the commit protocol. One-phase-standard and one-phase-delayed servers

[6] This can occur during a **Commit** by a single-process client with uncompleted asynchronous IPC requests, or by a multiprocess client with uncompleted synchronous requests.

[7] ω-requests include stateless requests (timeouts, polls), requests for user or device input, and the like. All requests to stateless servers (see Section 3.2) are ω-requests. Servers define their interface to allow ω-requests to be identified. Since requests are typed, this is implemented by defining special types for such requests.

receive their **end** message (terminating their participation) after the transaction commits. Rules 2 and 3 ensure that these servers do not issue any requests that could otherwise force the transaction to abort, after it has already committed.

4.3.2 *Cycles in the Transaction Graph.* It is possible for cycles to occur in the transaction graph, for example, when a server on node A (S_A) sends a request on behalf of transaction *T* to a server on node B (S_B), which itself requests S_C on node C, which requests S_B. In this case, TM_B has two superior, TM_A and TM_C. To accommodate this, each subordinate TM in a transaction distinguishes which of its superior TMs was the first to issue a request since the start of the transaction.[8] The subordinate TM initiates commit processing when this *first superior TM* sends a **vote** request. TM responds **vote-commit-read-only** to the **vote** requests of all other superior TMs, including new ones that appear during commit processing. When TM has collected votes from its local servers and all subordinates, it completes the first superior's **vote** request with its vote on the transaction.

In the above example, TM_A would send a **vote** request to TM_B, which would send a **vote** request to TM_C, which would send a **vote** request to TM_B. TM_B would respond **vote-commit-read-only** to TM_C, which would then (assuming the transaction was going to commit) respond **vote-commit** to TM_B, which would itself respond **vote-commit** to TM_A.

4.3.3 *New Requests after Becoming Prepared.* It is possible for new requests to arrive at a server after it has voted to commit (e.g., if server S_A calls already prepared server S_B). S_B can avoid atomicity problems in a rather heavy-handed way by refusing to process further requests (i.e., returning a bad return code), causing S_A to abort the transaction (S_A can not have voted). However, such is not our way. Instead, a prepared server that receives new work on behalf of a transaction is treated as a new participant. By Rule 1, TM allows the server to **re-vote** by sending another **vote** request to the server, which again becomes prepared and responds **vote-commit**. Here, if S_A and S_B are on different nodes, and if TM_B is already prepared, TM_A becomes the new "first" superior, and TM_B sends a **vote** request to S_B when it receives a **vote** request from TM_A.

It is possible that either TM_B or S_B will not be able to again become prepared, forcing it to respond **vote-abort**. The apparent violation of atomicity is resolved by the observation that the coordinator will not yet have decided to commit and will eventually see the **vote-abort**.

4.3.4 *Reappearance of a Forgotten Transaction.* Some systems [20] allow a node to unilaterally abort a transaction locally without informing any other nodes. If a transaction returns to a node that had locally aborted it, the transaction may be interpreted as a new one and subsequently committed. This will break atomicity, as some of the effects of the transaction will be permanent while some have evaporated with the local abort. The protocol described in [20] uses a system of time-stamps and low-water marks to preserve atomicity in such situations.

[8] Or since the most recent **vote** request (see Section 4.3.3).

In QuickSilver, a TM at a node can unilaterally abort a transaction and forget about it after informing its first superior TM. The effects of the transaction may be rolled back immediately by participants at or beneath that node. It is not necessary to force an abort record to the log, since any node failure prior to completion will cause the transaction to abort anyway. Our approach requires remembering the aborted transaction until the parent knows about its aborted status, but saves the extra bookkeeping of time-stamps and low-water marks associated with all work requests required by the protocol described in [20].

4.4 Coordinator Reliability

The coordinating TM is ordinarily the one at the transaction's birth-site. In the performance-critical case of a strictly local transaction, this is the correct choice. Most transactions are created by user workstations, which are the most likely to fail (e.g., when the user bumps the power switch or turns off the machine to go home). Coordinator failure during execution of the commit protocol for a transaction involving resources at remote recoverable servers can cause resources to be locked indefinitely.

QuickSilver uses two mechanisms to harden the coordinator. Both solutions—coordinator migration and coordinator replication—are cheaper and simpler than the Byzantine agreement protocol proposed by other researchers [24].

4.4.1 *Coordinator Migration.* At commit time, when the coordinator knows that a transaction has become distributed, it can designate a subordinate TM to take over as the coordinator. The topology of the transaction is changed to reflect the fact that the birth-site TM becomes a subordinate. Migration is used when the coordinator has only a single subordinate, in which case the subordinate is selected as the new coordinator. This corresponds to the common case of a program accessing files at a remote file server. Migration is accomplished by the coordinator (TM_A) first requesting votes from its local servers. After they respond, TM_A sends a special variant of the **vote** request to the subordinate (TM_B), naming it as the new coordinator, and specifying if TM_A needs to be included in phase two of the commit protocol. TM_B takes over the role of coordinator, requesting votes from its participating servers and subordinate TMs.

Migration tends to locate the coordinator where the transaction's shared, recoverable resources are (e.g., at a file server), which reduces the probability of a functioning server having to wait for a failed coordinator. When, as is often the case, TM_A has no two-phase participants in the transaction, coordinator migration also saves a remote IPC request. However, the migrated coordinator is still a single point of failure.

4.4.2 *Coordinator Replication.* For transactions in which the coordinator has multiple subordinates, QuickSilver allows the coordinator to be replicated to shorten the interval during which it is vulnerable to single-point failures. The basic idea is to select a subset of the subordinates as backup coordinators, to use a hierarchical two-phase commit protocol between the remainder of the subordinates and the coordinators, and to use a special protocol among the coordinators. In theory, one can use any number of replicas and any suitable protocol

(e.g., Byzantine agreement) among the replicas. QuickSilver uses a simple two-way replication algorithm.

A coordinator TM_A replicates itself by sending a special variant of the **vote** request to a subordinate TM_B. TM_B then sends a **vote** request to TM_A. This partitions the transaction graph into two blocks, one composed of coordinator TM_B and its subordinates, and one composed of coordinator TM_A and all its other subordinates. TM_A and TM_B regard each other as their standby. TM_A and TM_B then send **vote** requests to subordinates in their respective blocks, including in the request the name of both coordinators. The following describes the protocol from TM_A's standpoint; TM_B behaves likewise. When all TM_A's participants and subordinates have responded **vote-commit**, TM_A forces a **prepared** log record and then completes TM_B's **vote** request. When it is prepared *and* it has received the completion of its **vote** request to TM_B, it sends an **end-commit** request to TM_B. Upon receiving an **end-commit** request from TM_B, TM_A forces its **commit** record and sends **end-commit** requests to its subordinates. When these requests have been completed, TM_A completes TM_B's **end-commit** request. When TM_B has completed TM_A's **end-commit** request, TM_A writes its **end** record. If TM_B fails, then TM_A aborts if it has not yet sent **end-commit** to TM_B, otherwise it remains prepared. If a coordinator fails, its subordinates contact the standby to determine the outcome of the transaction.

The protocol blocks only if a failure occurs during the exchange of **ready** messages (an exceedingly short interval in practice). The cost is the **vote** and **end** requests from TM_B to TM_A, the **prepared** log record at TM_A, and the **commit** record at TM_B.

5. LOGGING AND RECOVERY

Each node's recovery manager contains a Log Manager (LM) component that is used by TM to write its commit protocol log records and is also used by other servers[9] that manage recoverable data. Providing a common log for use by all servers imposes conflicting goals of generality and efficiency on LM. If one were to port a significant subsystem, such as a database manager, to QuickSilver, LM is intended to be general enough to not force restructuring of the database's recovery architecture, and efficient enough to allow the database to run without significant performance penalty.

Of these two goals, efficiency was the simpler to achieve. For example, because of the use of a common log, servers can take advantage of TM's log forces to avoid doing their own during commit processing. Generality is more difficult, as even a single database manager or file system contains many storage structures, with different recovery techniques being most appropriate for each. Rather than trying to impose a fixed set of recovery techniques on such servers, LM offers a relatively low-level interface to an append-only log. This interface provides a core set of services, including restart analysis, efficient access methods, and archiving. On top of this interface, servers implement their own recovery

[9] The log may in fact be used by any recoverable program (e.g., long-running applications), but to simplify the text we will call any program that calls LM a "server".

algorithms and, in fact, drive their own recovery. This allows them to tailor their recovery techniques to those most appropriate for the data they maintain.

5.1 Log Manager Interface

The log consists of a large, contiguous address space subdivided into 512-byte *log blocks*. Each byte is addressable by a unique, 64-bit *log sequence number* (LSN). The log is formatted into *log records*. Each log record contains an abbreviated version of the *recovery name* used by servers to identify their log records, the *Tid*, and the server's data. Records may be of any length and can span any number of log blocks. Records from different transactions and different servers are freely intermixed in the log.

Before using the log, servers call LM to **identify** themselves, specifying their recovery name and the optional log services they require (see below). The server can then **read**, **write**, or **force** (synchronously write) records to the log. **Write** and **force** return the record's LSN.

A server can **read** records from the log in one of two ways: by providing the actual LSN or, more commonly, by opening a *scan* on the log via a logical *cursor*. A server can scan all its records, or just those of a particular transaction. LM returns the data, the *Tid*, and the status of the transaction (e.g., Prepared, Committed, Aborted). A server can read only valid records with its recovery name. To locate a starting point for recovery, servers are provided access to the *log restart area*. Servers typically save the LSN of a *log checkpoint* (see below) in the log restart area.

5.2 Log Operation and Services

LM formats log records received from servers into log blocks and buffers them. Buffered blocks are written to the nonvolatile *online log* either when buffer space is exhausted or when a server (or TM) calls **force**. The online log is structured as a circular array of blocks. Newly written blocks are written as the *head* of the log; each newly written log block overlays the oldest previously existing block. If that block still contains *live* data (i.e., data that is still needed by some server), it is copied to a log *archive* from which it can still be read, although perhaps at a performance penalty. Because the log is common to all servers, **force** causes all previously written records from all servers to be written to nonvolatile storage. TM exploits the common log to reduce the number of log forces during commit processing. When a server responds to TM's **vote** request, it specifies an LSN that must be forced before the server can enter the prepared state. Thus the log needs to be forced only once per transaction (by TM), regardless of the number of servers writing log records.

TM and each server have a distinct *log tail*, which is the oldest record they will need for crash recovery. When LM's newly written log records approach a server's log tail, LM asks the server at its option to take a *log checkpoint*.[10] In response, a server performs whatever actions are necessary (e.g., flushing buffers or copying log records) to move its log tail forward in the log and thus avoid having to access archived data during recovery.

[10] Log checkpoints are distinct from transaction checkpoints, described in Section 3.5.

When servers identify themselves to LM they define which optional log services they require. Dependencies between options are minimized so that, where possible, servers are not penalized for log services they do not use, nor by log services used by other servers. LM provides the following optional log services:

(1) **Backpointers on log records.** Servers that modify data in place need to replay their records for aborted transactions. This can be done efficiently by requesting LM to maintain backpointers on all log records written by that server for each transaction.

(2) **Block I/O access.** Servers that write large amounts of log data can call a set of library routines that allow them to preassign a contiguous range of log blocks, construct their own log records in these blocks, and write them as a unit, rather than calling LM for individual records. Since servers and LM are in separate address spaces, servers must explicitly write their preassigned blocks (i.e., they are not automatically written by TM log forces). Crashes can therefore create "holes" in the *physical* log. LM bounds the maximum contiguous range of preassigned blocks, and thus can recognize holes and skip over them during recovery. The holes will not affect the *logical* log of the client, which will always be contiguous.

(3) **Replicated logs.** Servers managing essential data may require the log to be replicated to guard against log media failure. LM supports log replication either locally or to remote disk servers.

(4) **Archived data.** Log blocks are archived when the online log wraps around on live data. This includes log records from inoperative servers (i.e., those that have crashed and not yet restarted), records from servers that do not support log checkpoints, and certain other records (e.g., records necessary for media recovery, records of long-running transactions). Except for a performance difference, the fact that a record is archived rather than in the online log is transparent to a server reading the record. The archive is stored in a compressed format so that only records containing live data are stored. The archive may be replicated, either locally or to remote archive servers.

5.3 Recovery

During recovery, LM scans the log starting with TM's log tail. This *analysis* pass determines the status (prepared, committed, aborted) of each transaction known to TM at the time of the crash, and builds pointers to the oldest and newest records for each such transaction written by each server. It also builds an *index*, used to support scans, that maps each server's log records to the blocks that contain them.

At the completion of the analysis pass, LM starts accepting *identify* requests from servers. At this point, servers begin their own recovery. In QuickSilver, servers drive their own recovery by scanning the log and/or by randomly addressing log records. Scans may retrieve all records written by the server or only the records written by a specific transaction. The server determines the number of passes over the log and the starting position and direction of each pass, and implements the recovery actions associated with the log records.

By remaining independent of the recovery protocol and by allowing servers to drive their own recovery, the QuickSilver recovery manager potentially incurs a higher cost during recovery than if it restricted the recovery protocols used by clients and drove recovery from the contents of the log. We attempt to minimize this cost in several ways. The index over the log maintained by LM allows it to read only the blocks that actually contain a server's data. The index, in association with the directional information associated with log scans, allows LM to prefetch data blocks in anticipation of the client's **read** requests. Also, the LM-maintained backpointers minimize the cost of backward scans. Finally, the results of the LM analysis pass are made available to servers. For some three pass-recovery protocols [14, 15, 33] the results of the TM's recovery can be used to simplify, or even to replace, the first pass of the protocol.

6. PERFORMANCE ANALYSIS

Table I summarizes the costs incurred by the QuickSilver recovery manager. There is one column for each of the four commit protocols, *One-Phase*, *Read Only*, *Two-Phase Volatile*, and *Two-Phase Recoverable*. The *Cost per Transaction* is a fixed overhead that is incurred regardless of the number of participants or the distribution. For rows in this section, and in the *Cost per Subordinate* section, the protocol column is selected as the maximum of that node's participant protocols and its subordinate protocols. The *Cost per Subordinate* rows show IPC requests between TMs on different nodes, and commit protocol log writes at the subordinate node. The protocols used between TMs are always two-phase.[11]

To allow comparison with other systems running on other hardware, Table II shows the cost of the base operating system functions used by the recovery manager. These (and all later benchmarks) were measured on RT-PC Model 25s (about 2 RISC mips) with 4 megabytes of memory, IBM RT-PC token ring adapters (4 megabit/sec transfer rate), and M70 70 megabyte disks (5 megabit/sec transfer rate). QuickSilver, as well as all server and client processes, were compiled with the PL.8 compiler [2]. TM uses short IPC messages, and LM uses streamed writes. The table entries for 1K-byte IPC messages and random-access disk I/O will be used in later benchmarks. Remote IPC performance was measured on a lightly loaded network, but because of the characteristics of the token ring, performance does not degrade significantly until network load exceeds 50 percent.

Given these base system and I/O performance numbers, a series of benchmarks was run to determine the actual overhead of the transaction and logging mechanism. We used a set of benchmarks similar to those reported for Camelot in [36], which, in addition to providing a way of determining the recovery management overhead, allows a direct comparison of QuickSilver's performance to that of at least one other system under a similar set of conditions.

All benchmarks were run on otherwise unloaded machines, with an unloaded network, a simplex log, and unreplicated coordinators. Each number in Table III

[11] Table I describes the case where the coordinator is not being replicated. The cost per transaction increases by two remote IPC requests and two force log writes when the coordinator is replicated (see Section 4.4.2).

Table I. Transaction Management Algorithmic Costs

	Transaction Protocol			
Cost per transaction	One phase	Read only	Two-phase volatile	Two-phase recoverable
Begin transaction	1 Local IPC	1 Local IPC	1 Local IPC	1 Local IPC
Commit/abort transaction	1 Local IPC	1 Local IPC	1 Local IPC	1 Local IPC
Log commit/abort record	0	0	0	1 Log force
Log end record	0	0	0	1 Log write
Cost per participant				
Request vote	0	1 Local IPC	1 Local IPC	1 Local IPC
Commit/abort transaction	1 Local IPC	0	1 Local IPC	1 Local IPC
Cost per subordinate				
Request vote	—	1 Remote IPC	1 Remote IPC	1 Remote IPC
Commit/abort transaction	—	0	1 Remote IPC	1 Remote IPC
Log prepare record	—	0	0	1 Log force
Log commit/abort record	—	0	0	1 Log force
Log end record	—	0	0	1 Log write

Table II. Primitive Operation Times in msecs.

Primitive operation	Time
Local 32-byte IPC	.66
Local 1K-byte IPC	1.16
Remote 32-byte IPC	9.0
Remote 1K-byte IPC	16.0
Average 512-byte streamed raw disk I/O, including cylinder steps	2.3
Random-access 4096 byte I/O, read or write	37.5

is the per-transaction average over 4 runs, each run consisting of a batch of 4096 32-byte transactions or 512 1K-byte transactions. The write benchmarks caused log checkpoints, and the time for these are included in the averages. As in [36], the benchmark transactions were run serially from a single application, and all service requests were synchronous, as the goal was to measure transaction management overhead as opposed to response time or throughput.

The following benchmarks were run:

(1) Transactions on 1, 2, and 3 local servers that read or write one 32-byte record. These demonstrate the basic overhead of local read and write transactions, and the incremental cost of involving additional servers.

(2) Transactions on 1, 2, and 3 local servers that read or write ten 32-byte records (as ten separate synchronous requests). These allow computing the incremental costs of additional operations on servers from an existing transaction, which then allows computing the local per-transaction overhead, including that of log forces for write transactions.

Table III. Benchmarks on 1–4 RT-PCs, msecs/transaction

Transaction benchmarks	1 Server	2 Servers	3 Servers
Local reads			
1 32-byte read/server	6.1	8.7	11.2
10 32-byte reads/server	14.1	24.7	35.2
1 1K-byte read/server	6.6	9.6	12.6
10 1K-byte reads/server	18.7	33.8	48.9
Local writes			
1 32-byte write/server	41.7	41.7	42.4
10 32-byte writes/server	58.4	92	125
1 1K-byte write/server	41.7	50.8	66.9
10 1K-byte writes/server	119	181	239
Remote reads			
1 32-byte read/server	31.9	45.5	58.8
10 32-byte reads/server	121	224	329
1 1K-byte read/server	38.1	57.9	77.5
10 1K-byte reads/server	183	348	515
Remote writes			
1 32-byte write/server	77	101	122
10 32-byte writes/server	201	325	447
1 1K-byte write/server	80	124	152
10 1K-byte writes/server	335	533	725

(3) All of the above with 1K-byte records.

(4) All of the above with the application on one node and each data server on a separate node. This demonstrates the additional overhead for distributed server requests and committing distributed transactions.

The numbers in Table III were used to derive the transaction management costs shown in Table IV. For example, for local read-only transactions there is a fixed overhead of 3.5 msec. and a per-server overhead of 1.7 msec. Comparing these numbers with the numbers that can be derived from the primitive operation times from Table II and the algorithmic operation costs from Table I allows one to get a rough idea of the execution time of TM and LM. For example, simple read transactions require two IPC requests (**Begin** and **Commit**) plus one IPC request (**Vote**) for each of the n participating servers. This adds up to $1.32 + .66n$, so the execution time in TM is approximately $2.18 + 1.04n$ msec.

The equations for read transactions in Table IV closely match the benchmark data points. The write transactions were more difficult to measure accurately. Benchmark transactions were run serially, and were the only transactions running in the system. Because LM physically writes its log contiguously on the disk, a complete revolution is missed between transactions, and transaction execution is effectively synchronized to the disk rotation rate. Transactions arriving at random times would see faster response time. It is also interesting to note that if other log activity were occurring that allowed the log to "keep up" with disk rotation, response times for small transactions could be much lower than those observed in the benchmarks. To allow the CPU overhead of write transactions to be observed independently of the effects of disk rotation, Table V shows the results of repeating the local write benchmark with the LM disk driver call changed from "write" to "no-op".

Table IV. Approximate Elapsed Times in msecs of Various QuickSilver Functions on the RT-PC

QuickSilver function	Time
Cost/transaction for n servers	
Local read-only	$3.5 + 1.7n$
Local write	$31.0 + 4.2n$
Remote read-only	$18.5 + 3.6n$
Remote write	$53.0 + 7.3n$
Cost/read operation	
Local 32-byte	0.89
Local 1K-byte	1.35
Remote 32-byte	9.9
Remote 1K-byte	16.1
Cost/write operation	
Local 32-byte	2.8
Local 1K-byte	7.2
Remote 32-byte	13.8
Remote 1K-byte	28.3

Table V. Local Write Transaction CPU Cost

Transaction benchmarks	1 Server	2 Servers	3 Servers
1 32-byte write/server	19.5	25.7	31.7
10 32-byte writes/server	43.1	72.7	102
1 1K-byte write/server	20.5	28.4	35.4
10 1K-byte writes/server	57.4	103	148

QuickSilver function	Time
Cost/transaction for n servers	
Local write	$13.2 + 4.2n$
Cost/write operation	
Local 32-byte	2.6
Local 1K-byte	4.1

Table VI. Remote Read Transactions, 1 32-Byte Asynchronous Read/Server, msecs/transaction

Number of servers	Time (change vs. synch)
1 Server	32.0 (+0.3%)
2 Servers	37.5 (−17.6%)
3 Servers	42.9 (−27.0%)

Finally, it is important to point out that the raw performance numbers are intended to be used to derive the per-transaction operational costs. They do not exploit possible parallelism, and thus are not an indication of potential throughput. To illustrate this, we repeated the benchmark for 32-byte remote read transactions, but changed it to make asynchronous requests to the servers. The results, with percent changes from Table III, are shown in Table VI. The slight decrease in time for one node is due to the extra kernel calls to wait on the group of requests. The fact that execution time grows with the number of servers shows that parallelism is not perfect; all messages go to or from the client's node, so the network and the client node's Communication Manager act as a bottleneck.

7. RELATED WORK

Several systems described in the literature use transaction-based recovery as a lower-level component of a higher-level entity such as a file system or database. For example, System R and R* [15, 19] implement relational databases that support atomic transactions. Locus [26, 38] offers a transactional file system.

Other systems, such as Argus [21], Eden [30], Clouds [1], CPR [8], and Avalon [16] implement programming languages that include constructs for recoverable data objects built on top of a lower-level transaction-based recoverable storage manager.

Camelot [36] (and its precursor TABS [34]) integrates transaction-based recovery and write-ahead logging with virtual memory in a manner similar to CPR, but uses software rather than special-purpose hardware to control access to recoverable storage. Camelot offers recoverable storage in the context of a standard programming language (C) via a macro package and library routines. Camelot macros hide logging, recovery, and commit processing from servers that manage recoverable resources. Applications start and end transactions and call servers via Camelot macros that generate Mach RPC calls. Unlike Argus and CPR (which support redo logging), Camelot implements both redo and undo/redo logging. It also allows a choice of blocking or nonblocking commit protocols, and supports nested transactions [25] in a manner similar to Argus.

The V-System [9] implements transactions on top of its process group facility [10, 11]. In V, transactions are implemented via a transaction library running as part of the client process, a transaction log server, and data servers that manage recoverable objects. Each transaction is represented by a process group. A client calls the (possibly replicated) log server to create a transaction, adds each transactional server it calls to the transaction's process group, and passes the transaction ID as a parameter to the server. The client multicasts **prepare-to-commit** messages to the group, and when all respond affirmatively, calls the log manager to commit the transaction.

While there is much in QuickSilver recovery management that is similar to the aforementioned systems, QuickSilver differs from them in several significant ways. The basic difference is the use of transactions as a unified recovery mechanism for both volatile and recoverable resources. This motivated the lightweight extensions to the commit protocol and is reflected in the low overhead exhibited in the benchmarks. This also led to the fact that QuickSilver directly exposes the recovery management primitives at a lower level than most comparable systems.[12] In particular, servers implement their own recoverable storage, choose their own log recovery algorithms, and drive their own log recovery. In addition to being more flexible and potentially more efficient for servers developed especially for QuickSilver, this approach also simplifies porting recoverable services developed for other systems to QuickSilver by mapping their existing recovery algorithms onto the corresponding QuickSilver primitives.

Another important difference is QuickSilver's integration of recovery management into IPC. There is no special "server call" mechanism for recoverable

[12] Camelot offers a "primitive interface" that allows servers more direct control of their storage, but encourages using the higher-level library.

servers as in Camelot. This, plus the QuickSilver notion of system-created "default" transactions, allows client programs written in conventional programming languages to be completely unaware of the recovery mechanism and still behave atomically. This greatly facilitates porting programs such as file-processing applications to QuickSilver. IPC automatically tracks server participation in transactions, which eliminates the need for system calls to add servers to transactions as in V [10], and allows implementing the "re-vote" mechanism, which itself eliminates the need for distinct "close" calls to servers to quiesce them prior to initiating commit processing.

8. STATUS AND CONCLUSIONS

QuickSilver is installed and running in daily production use on 47 IBM RT-PC's in the computer science department at IBM Almaden Research Center and at other IBM locations. In addition to the QuickSilver group, several other research projects are using QuickSilver as an environment to develop applications and network-based services. The recovery manager has been implemented and is being used as the recovery mechanism for all QuickSilver servers.

The benchmarks described above showed that the recovery management overhead is small. In the simple, normal case (i.e., the transaction commits, no failures occur, and no re-vote is required), the recovery manager requires a minimal number of messages, and CPU overhead is very small. Experience with the system has confirmed that recovery management overhead is negligible and not perceptible to users. We believe this shows that the mechanism is efficient enough to be used for servers with very stringent performance demands.

We were concerned when we started the design of the recovery manager that there would be exactly two types of users: simple servers, like the window manager that just need a completion notification mechanism; and the file system, which needs distributed two-phase commit in its full glory. In fact, we have found transactions to be useful in a variety of applications, and have found the ability to decouple commit coordination from logging and to use them individually to be valuable as well. We mentioned several such cases: the replicated name server, which uses commit coordination but not logging, and checkpointable applications, which use logging but not the commit protocol. We are experimenting with other applications, including a distributed messaging facility and a mail store-and-forward system.

Development of the QuickSilver Distributed File Services (DFS) [7] confirmed the use of server-defined recovery algorithms and server-driven recovery. The approach taken by other systems of embedding all recovery processing within the recovery manager simplifies programming servers. However, DFS pointed out several shortcomings in this style of transparent recovery. Certain operations on file system metadata require operation logging (e.g., B-tree inserts that may provoke splitting the tree are undone/redone operationally to avoid locking large subtrees). The DFS storage allocator, which uses a bitmap, implements its own value logging and concurrency at the bit level, a granularity not to our knowledge supported by any of the aforementioned recovery managers. Transaction checkpoints were motivated by various DFS metadata updates (e.g., B-tree splits) that are done on behalf of DFS internal transactions rather than client transactions

to maximize concurrency and eliminate unnecessary undo operations after client-transaction aborts. Checkpoints allow the updates to be logged and recovered without the overhead of starting a new transaction for each one. Experience seems to show that the log index, prefetch during scans, and a reasonable amount of log buffer memory provide more than adequate recovery performance.

We intend to pursue development of the QuickSilver recovery manager in the following areas:

(1) **Deadlock Detection.** As mentioned earlier, no work has been done on the Deadlock Detection component of the recovery manager. We anticipate beginning this work shortly.

(2) **High-Performance Servers.** The "block access" log interface reduces the number of calls to the log manager, but causes sparser utilization of log blocks and more log block writes. Considerably more performance analysis is necessary to evaluate the benefit of block access for servers like the file system that potentially log large amounts of data.

(3) **Nested Transactions.** QuickSilver presently does not include a nested transaction mechanism. The utility of a mechanism such as that proposed by Moss [25] is clear, and we intend to investigate implementing one.

(4) **Recoverable Object Managers.** QuickSilver's recovery manager is intended primarily as a tool for use by low-level servers, and as such trades ease of use to gain flexibility and efficiency. However, we recognize the merit of systems like Camelot and Argus that make it easy to define and use recoverable objects. It is relatively straightforward to implement recoverable object managers on top of the QuickSilver recovery primitives. We intend to explore a language-directed facility for defining and using recoverable objects, perhaps in the context of a language such as C++.

ACKNOWLEDGMENTS

We would like to thank Jim Wyllie and Luis-Felipe Cabrera, whose work on the architecture and implementation of the QuickSilver Distributed File Services has helped to drive the design of the recovery manager and has provided a testbed to debug it.

REFERENCES

1. ALLCHIN, J. E., AND MCKENDRY, M. S. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 31–44.
2. AUSLANDER, M., AND HOPKINS, M. An overview of the PL.8 compiler. In *SIGPLAN '82 Symposium on Compiler Writing* (Boston, Mass., June 1982). ACM, New York, 1982.
3. BARON, R. V., RASHID, R. F., SIEGEL, E. H., TEVANIAN, A., JR., AND YOUNG, M. W. MACH-1: A multiprocessor oriented operating system and environment. In *New Computing Environments: Parallel, Vector, and Systolic*, SIAM, 1986, 80–89.
4. BARTLETT, J. A NonStop kernel. In *ACM Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif. Dec. 1981). ACM, New York, 1981, 22–30.
5. BIRMAN, K. P. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 79–86.

6. BORR, A. J. Transaction monitoring in Encompass: Reliable distributed transaction processing. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, Sept. 1981), IEEE, New York, 1981, 155–165.
7. CABRERA, L. F., AND WYLLIE, J. C. QuickSilver distributed file services: An architecture for horizontal growth. IBM Res. Rep. RJ5578, Feb. 1987.
8. CHANG, A., AND MERGEN, M. F. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.* This issue, 28–50.
9. CHERITON, D. R. The V kernel: a software base for distributed systems. *IEEE Softw. 1*, 2 (April 1984), 19–42.
10. CHERITON, D. R. Fault-tolerant transaction management in a workstation cluster. Unpublished.
11. CHERITON, D. R., AND ZWAENEPOEL, W. Distributed process groups in the V kernel. *ACM Trans. Comput. Syst. 3*, 2 (May 1985), 77–107.
12. COOPER, E. C. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 63–78.
13. CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Res. Rep. RJ5244, IBM, San Jose, Calif., July 1986.
14. GRAY, J. N. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller, Eds. Springer-Verlag, New York, 1978, 393–481. Also available as IBM Res. Rep. RJ2188, IBM Almaden Research Center, San Jose, CA 95120.
15. GRAY, J. N., MCJONES, P., BLASGEN, M. W., LORIE, R. A., PRICE, T. G., PUTZOLU, G. F., AND TRAIGER, I. L. The recovery manager of the System R database manager. *Comput. Surv. 13*, 2 (June 1981), 223–242.
16. HERLIHY, M. P., AND WING, J. M. Avalon: Language support for reliable distributed systems. Tech. Rep. CMU-CS-86-167, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., Nov. 1986.
17. INTERNATIONAL BUSINESS MACHINES. Systems Network Architecture Transaction: Programmer's Reference Manual for LU Type 6.2, IBM Corporation GC30-3084.
18. LAMPSON, B. W. Atomic transactions. In *Distributed Systems—Architecture and Implementation*. Springer-Verlag, New York, 1981, 246–264.
19. LINDSAY, B., HAAS, L., MOHAN, C., WILMS, P., AND YOST, R. Computation and communication in R*: A distributed database manager. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 1983). ACM, New York, 1983, 1–10. Also available as IBM Res. Rep. RJ3740, IBM, San Jose, Calif., Jan. 1983.
20. LINDSAY, B. G., SELINGER, P. G., GALTIERI, C., GRAY, J. N., LORIE, R. A., PRICE, T. G., PUTZOLU, F., TRAIGER, I. L., AND WADE, B. W. Single and multi-site recovery facilities. In *Distributed Data Bases*, I. W. Draffan and F. Poole, Eds. Cambridge University Press, Cambridge, UK, 1980. Also available as Notes on Distributed Databases, IBM Res. Rep. RJ2571, IBM, San Jose, Calif., July 1979, 44–50.
21. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 381–404.
22. LYON, B., AND SAGER, G. Overview of the SUN network file system. SUN Microsystems, Inc., Mountain View, Calif., Jan. 1985, 1–8.
23. MOHAN, C., LINDSAY, B., AND OBERMARCK, R. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986), 378–396. Also available as IBM Res. Rep. RJ5037, IBM, San Jose, Calif., Feb. 1986.
24. MOHAN, C., STRONG, H. R., AND FINKELSTEIN, S. Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (Montreal, Aug. 1983). ACM, New York, 1983, 89–103. Also IBM Res. Rep. RJ3882.
25. MOSS, E. B. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, Mass., 1985.
26. MÜLLER, E. T., MOORE, J. D., AND POPEK, G. J. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating System Principles* (Bretton Woods, N.H., Oct. 1983). ACM, New York, 1983, 71–89.
27. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June 1982), 187–208.

28. OKI, B., LISKOV, B., AND SCHEIFLER, R. Reliable object storage to support atomic actions. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 147–159.
29. POPEK, G., WALKER, B., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN, G., AND THIEL, G. LOCUS: A network transparent high reliability distributed system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 1981). ACM, New York, 1981, 169–177.
30. PU, C., NOE, J. D., AND PROUDFOOT, A. Regeneration of replicated objects: A technique and its Eden implementation. In *Proceedings of the 2nd International Conference on Data Engineering*, (Los Angeles, Feb. 1986). IEEE Press, New York, 1986, 175–187.
31. RASHID, R., AND ROBERTSON, G. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 1981). ACM, New York, 1981, 64–75.
32. REED, D., AND SVOBODOVA, L. SWALLOW: A distributed data storage system for a local network. In *Networks for Computer Communications*, North-Holland, Amsterdam, 1981, 355–373.
33. SCHWARZ, P. M. Transactions on Typed Objects. Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1984. Available as CMU Tech. Rep. CMU-CS-84-166.
34. SPECTOR, A. Z., BUTCHER, J., DANIELS, D. S., DUCHAMP, D. J., EPPINGER, J. L., FINEMAN, C. E., HEDDAYA, A., AND SCHWARZ, P. M. Support for distributed transactions in the TABS prototype. *IEEE Trans. Softw. Eng. SE-11*, 6 (June 1985), 520–530.
35. SPECTOR, A. Z., DANIELS, D., DUCHAMP, D., EPPINGER, J., AND PAUSCH, R. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985. ACM, New York, 1985, 127–146.
36. SPECTOR, A., ET AL. Camelot: A distributed transaction facility for Mach and the internet—an interim report. Tech. Rep. CMU-CS-87-129, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., June 1987.
37. STONEBRAKER, M. Operating systems support for database management. *Commun. ACM 24*, 7 (July 1981), 412–418.
38. WEINSTEIN, M. J., PAGE, T. W., LIVEZEY, B. K., AND POPEK, G. J. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1985). ACM, New York, 1985, 115–126.

Received May 1987; revised May 1987; accepted September 1987

The Implementation of An Integrated Concurrency Control and Recovery Scheme

Arvola Chan
Stephen Fox
Wen-Te K. Lin
Anil Nori
Daniel R. Ries

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Abstract

This paper describes the implementation level design of an integrated concurrency control and recovery scheme based on the maintenance of multiple versions of data objects in a database. The concurrency control mechanism enhances parallelism by eliminating interference between retrieval and update transactions. The recovery mechanism permits efficient transaction and system recovery by keeping before-images of data objects at the page (block) level. This paper addresses the key technical problems in the implementation of such an integrated scheme. We present an efficient garbage collection algorithm for reclaiming storage space used by old versions of data objects that will no longer be accessed. We also propose an on-line backup algorithm that will permit the backup procedure to run in parallel with regular transactions. This integrated concurrency control and recovery scheme is being implemented in the LDM: the local database manager component of a distributed database management system, now being developed by Computer Corporation of America, that will support the ADAPLEX database application programming language [Chan81a, Smith81].

This research was jointly supported by the Defense Advanced Research Projects Agency of the Department of Defense and the Naval Electronic Systems Command under Contract Number N00039-80-C-0402. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Naval Electronic Systems Command, or the U. S. Govern-

(1) Ada is a trademark of the Department of Defense (Ada Joint Program Office).

1. INTRODUCTION

The Computer Corporation of America is currently developing a high-performance, distributed database management system that is compatible with the programming language Ada.(1) This system will support the integrated database programming language ADAPLEX, which is the result of embedding the database sublanguage DAPLEX [Shipman81] in Ada [Dod80]. Two versions of the DBMS are being planned. A centralized DBMS, called the Local Database Manager (LDM), is designed for high performance as a stand-alone system. A distributed DBMS, called the Distributed Database Manager (DDM), will enable multiple LDMs to be interconnected in a computer network in order to provide rapid access to data by users who are geographically separated. This paper describes our implementation of an integrated concurrency control and recovery scheme (which is based on the maintenance of multiple versions of data objects) in the LDM. A similar scheme has previously been used in Prime's Database Management System [DuBourdieu82]. Our principal contributions in this paper include an algorithm for efficiently reusing space occupied by old versions of data objects that will no longer be accessed, and an algorithm for the saving of a consistent state of the database on backup storage (for media recovery) while other transactions continue to update the database. Section 2 provides an overview of the integrated scheme which is based on the maintenance of multiple versions of data objects. Section 3 details our solution to the garbage collection problem in the version pool in which old versions of data objects are maintained. Section 4 describes recovery management in the LDM and presents an on-line backup algorithm that can be used in conjunction with the maintenance of a recovery log to support recovery from hard crashes.

2. OVERVIEW OF INTEGRATED SCHEME

We are taking an integrated approach to solving the related problems of synchronizing concurrent transactions and of recovering from transaction and system failures in the LDM. This approach takes advantage of the presence of old versions of data objects traditionally used only

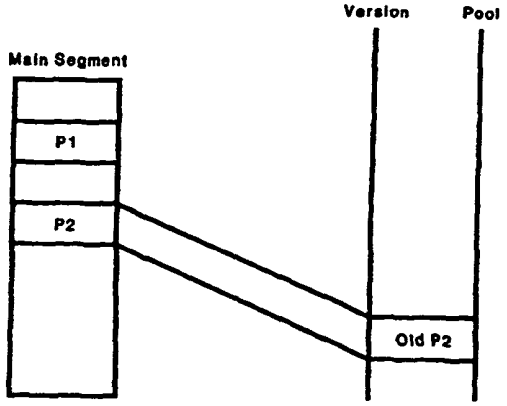
2.1 Potential Concurrency Enhancement

The standard approach to concurrency control in a centralized environment is through the use of two-phased locking [Eswaran76, Gray78]. Two-phased locking results in a serializable execution of any pair of conflicting transactions. In order to support recovery from transaction and system failures, the two-phased locking systems cannot simply overwrite the old values (versions) of data objects. These old versions are typically copied to a recovery log before any in-place update is performed. In the event of failures, data objects that have been updated by incomplete transactions are restored to their previous consistent state, using the appropriate old versions in the recovery log.

It has been observed that these old versions of data objects are under-utilized in contemporary database systems [Bayer80, Stearns81]. In addition to supporting the recovery mechanism, they can be exploited in order to improve the degree of concurrency achievable in the system. The scheme presented in [Bayer80] makes use of a single "before value" to improve concurrency. The scheme we are considering bears more resemblance to that discussed in [Stearns81] in that we exploit the presence of multiple old versions of a data object to eliminate read-write conflicts. It is different from the more general multi-version scheme proposed in [Reed78] in that a retrieval transaction is not allowed to ask for an arbitrary time slice of the database. Thus, we do not have the problem of the database growing in size indefinitely.

The mutual exclusion of concurrent retrievals and updates that operate on the same conceptual data object is unnecessarily restrictive in concurrency control schemes based on two-phased locking. With appropriate management of the different versions of each data object, it should be possible for an update transaction to modify a data object (i.e., create a new version for it) while other retrieval transactions are granted the privilege of reading other older versions of the same object.

Consider the example shown in Figure 2.1. Transaction T_1 needs to read logical pages P_1 and P_2, while transaction T_2 needs to write logical page P_2. If transaction T_1 already has read page P_1, and transaction T_2, instead of simply overwriting page P_2, places the existing version, old P_2, in the version pool, then it is still possible for transaction T_1 to obtain a consistent snapshot of the database by reading page version old P_2. In a two phase locking scheme, T_1 would have been blocked when it requested a read lock on page P_2 since T_2 had already acquired a write lock on page P_2. In our scheme, T_1 can proceed concurrently with T_2, instead of having to wait for T_2 to complete.



T1 : Reads P1

T2 : Updates P2 forcing old P2 to the version pool

T1 : Trys to read P2 and is presented with 'old' P2 since T2 was not complete when T1 started

Figure 2.1: Conflicting Reading and Update Transactions

2.2 An Integrated Approach

Exploiting the availability of multiple versions of data objects in order to improve concurrency requires the solution of a number of technical problems, including:

- Efficiently locating old versions of data objects.
- Determining the appropriate versions of different objects for a transaction to read.
- Reclaiming space occupied by aged object versions.

Briefly, our solutions to these problems involve:

- Chaining of versions of data objects in reverse chronological order.
- Labeling of object versions with identifiers of their creating transactions, and requiring that retrieval transactions read only object versions created by transactions that have completed before the time of their initiation. Transactions that both read and write always use (and lock) the latest version of an object.
- Storing of old versions of data objects in a common ring buffer in order to permit automatic garbage collection.

Our entire integrated scheme is described in more detail below. We assume that the database consists of a number of segments (files). Each segment is a collection of consecutively numbered pages. These pages are treated as units of data objects in our scheme. A common, fixed size version pool, which is physically separate from the segments proper, is maintained. The segments proper and the version pool are both stored on disk. The version pool is

used to store the old versions of pages from different segments. The various versions of a segment page (one in the segment proper, and zero or more in the version pool) are linked together in reverse chronological order.(2) In order to distinguish different versions of the same page, each version is labelled with the identifier of the transaction that created it.

The synchronization protocol requires that update transactions perform two-phased locking on the set of pages that are to be read or written. That is, only a single update transaction can be granted the exclusive privilege of creating a new version for any data object at any point in time, and the update transaction must also base its updates on the latest consistent version of the database. Deadlocks between update transactions are detected and resolved by a deadlock detector that runs periodically.

Read-only (retrieval) transactions, on the other hand, do not set locks. Instead, each of these transactions can obtain a consistent (but not necessarily most up-to-date) view of the database by following the version chain for any given logical page, and by reading only the first version encountered that has been created by an update transaction with a completion time earlier than the time of its own initiation. Thus, read-only transactions do not cause synchronization delays on update transactions, and vice versa.

The implementation of this scheme requires the maintenance of a completed transaction list. Transactions are assigned unique transaction numbers in chronological order. At the initiation of a read-only transaction, the current completed transaction list is consulted. The read-only transaction can see updates only from this set of transactions. Before an update transaction performs any in-place update on a page, the old version of the page is first copied to the version pool, and the version chain is appropriately updated.(3) Thus, a read-only transaction which has started earlier can simply follow the version chain to locate the appropriate version to read.

2.3 Garbage Collection

Potentially serious problems with a multi-version scheme can arise due to the difficulty of garbage collection. The issues here involve:

- When to garbage collect. If garbage collection and reuse of a particular page version is performed too early, two undesirable situations can arise:

  1. An ongoing update transaction may not be abortable because the before image of a data object which it has overwritten is no longer available.

  2. An ongoing read-only transaction may not be able to complete because one of its desired page versions has been garbage collected from the pool.

  On the other hand, if garbage collection is performed too late, then a lot of wasted space will result and a larger amount of space has to be set aside for the version pool.

- How to maintain acceptable performance. There are three potential performance bottlenecks:

  1. An on-going update transaction should be able to determine if a version pool page is reusable without having to examine its contents.

  2. A garbage collector should be able to declare a page "free" for reuse without having to follow a linked list on disk. The multiple disk reads would be prohibitively expensive.

  3. Once a page has been declared "free" for reuse, a garbage collector should not have to locate and/or update a pointer on the predecessor disk page.

Our solution to these difficult garbage collection problems is detailed in the following section.

3. VERSION POOL MANAGEMENT

We will refer to those old page versions (which will not be needed for undoing incomplete update transactions or for completing ongoing read-only transactions) as dead page versions. To automatically reclaim space used by dead page versions, the version pool is used as a ring buffer. That is, a sequentially incremented pointer is used to point to the next slot in the version pool that should be considered for reuse. Our strategy for determining if a particular slot is reusable is designed to:

- Avoid having to update links in version chains.
- Reuse only slots occupied by dead page versions.
- Avoid having to read the current contents of a slot to determine its reuseability.

Notice that if we can completely solve the second problem of reusing only slots occupied by dead page versions, then we also will have solved the first problem of eliminating link updates. No active

---

(2) The number of old versions of a data object that are present in the version pool will depend on the frequency of updates to that object and on the size of the version pool.

(3) An important reason for performing in-place update is to retain the clustering property of the most up-to-date versions of consecutively numbered segment pages.

transaction would follow a pointer to a dead page, so that no pointer need be updated. However, in some cases, it may be necessary to reuse a not yet dead page. In these controllable cases, it is necessary to determine if a slot in the version pool actually contains a desired page in a given version chain. We make such a determination by using a self-identification scheme for page versions in both the version pool and in the segments proper. That is, each page version is labelled with its segment number, logical page number, and the identifier of the transaction that has created it. The page identifier in a pointed-to slot in the version pool is examined to determine if it actually contains a version of a desired page. In addition, to avoid the possibility of an inappropriate old version of the desired page being read, the identifier of the transaction that has caused a page version to be copied to the version pool is also stored. We will call this the overwriting transaction identifier. In following a version chain, the overwriting transaction identifier in a pointed-to slot is required to match the creating transaction identifier of the version from which the "next" pointer is obtained.

3.1 Data Structures

The data structures used to support pointer validation and space management are further described in the next three subsections.

3.1.1 Page Header

As stated before, each data object is equated with a page in a segment. Therefore each data object can be uniquely identified by a segment number, along with a logical page number within that segment. A version header is stored with each page version. It has the following components:

- Page Identifier. This uniquely identifies the page (segment number plus page number) in question. Logically, this information is needed only in headers for page versions stored in the version pool.
- Creating Transaction Identifier. This identifies the update transaction that has created the page version.(4) A read-only transaction checks this information against its associated completed transaction list while traversing a version chain in order to determine if a given page version is one that it should see.
- Overwriting Transaction Identifier. This identifies the transaction that has created the next newer version, overwriting a copy of this version in the segment proper and causing it to be copied to the version pool. This information, along with the page identifier, is used to validate the fact that a pointed-to "next" older version is still in the version pool.
- Next Version Pointer. This points to the next older version of the same page in the version pool and is used by read-only transactions to traverse a version chain in reverse chronological order. It is simply represented as an offset within the pool.

3.1.2 Transaction Descriptor

The transaction descriptor stores a variety of information needed for concurrency control and recovery management. The information for each transaction that is relevant for version pool management includes:

- Transaction Type. This distinguishes read-only transactions from update transactions.
- Version Pool List (VP LIST). This keeps track of the collection of slots in the version pool used by an update transaction for storing before page images.
- Read Range Lower Bound (RRLB). This keeps track of the lower bound of a range of slots in the version pool that may contain page versions that may be read by a read-only transaction.
- Update Range Lower Bound (URLB). This keeps track of the lower bound of a range of slots in the version pool used by an update transaction for storing before page images.

3.1.3 Completed Transaction List

Transactions are assigned monotonically increasing transaction numbers. A transaction is said to be completed if it has been either committed or aborted. A list of all completed transactions is maintained in both main memory and in secure storage. The completed transaction list in main memory is consulted at the initiation of each read-only transaction to determine the set of transactions whose updates it should see. (Notice that versions created by an aborted transaction would not have appeared in the version chains.) The completed transaction list in secure storage is used during system recovery to identify the set of incomplete update transactions whose results should be backed out. The completed transaction list is represented by a base transaction number, along with a bit map, to signify completed transactions that have larger transaction numbers. Periodically, or at system quiescent points, the base transaction number is set to be equal to the largest completed transaction number, and the bit map is compacted correspondingly.

(4) Each page version, when initially created, is stored in the segment proper. It migrates to the version pool when a later version is created.

3.2 Page Replacement Algorithm

The version pool, of size VP_SIZE, is organized as a large ring buffer with slots numbered 0, ..., VP_SIZE - 1. Pages within this buffer normally are allocated in a sequential fashion. Associated with each page is one of three possible allocation statuses.

1. Free. A free page can be reused without affecting any ongoing transaction.

2. Potentially active. A potentially active page might be read by an ongoing read-only transaction. Such a page can be reused, but it might result in the abortion of a read-only transaction.

3. Active. An active page has been written by an ongoing update transaction. Such a page cannot be reused.

The algorithm for reusing slots in the version pool is designed to:

- Reuse only slots not needed for rolling back incomplete update transactions.
- When possible, avoid using slots potentially needed by an ongoing read-only transaction.
- Avoid having to read the contents of a slot before reusing it.

Whether a version pool page is active or not can be determined from information stored in the page header, along with information stored in the completed transaction list. However, use of this information would imply that the version pool space allocation algorithm must read each page before deciding whether or not the page can be reused. Alternatively, we could examine the version pool lists (VP_LIST) associated with ongoing update transactions and simply select a slot that is not on any of these lists.

Rather than using either of these two approaches we maintain sliding ranges of version pool pages that are associated with active transactions. These ranges are illustrated in Figure 3.1. Recall that the version pool is maintained as a ring buffer. For the purpose of simplifying this discussion, assume that the version pool is unlimited in size and that the first and last pointers continue to grow. Actually, the growth is modulo the version pool size. We require only that the last pointer not overtake either of the "first" pointers. We first describe the two ranges and what can be assumed about pages in and out of these ranges. We next describe the "normal" effects on these ranges when pages are reused and transactions are begun and committed. Finally, we describe the operations of the version pool when the version pool becomes "full", i.e., "the last" pointer attempts to overtake the first.

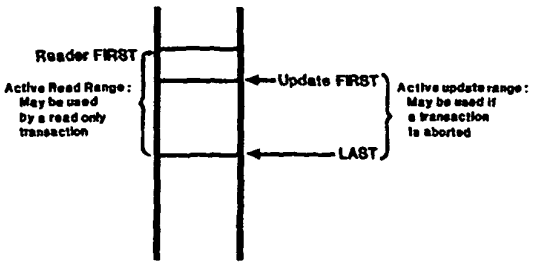


Figure 3.1: Version Pool Pages that may be used

The range of pages between a first pointer and the last pointer represents the pages in the version pool that may have to be used. The pages between update_first and last are the pages that may have been written by active update transactions. These are the only pages that would be needed to restore the database segment pages if a transaction were aborted. The pages between reader_first and last are the pages that may be read by a read-only transaction. In fact these are the only pages in the version pool that a read-only transaction would ever need to read.

We now describe how these pointers are updated during transaction execution when the version pool is not full.(5) We have three cases:

1. Transaction initiation. This action will not actually change the ranges. Instead, a read-only transaction, when it initiates, records the current value of the update_first pointer. This value is called the read range lower bound for this transaction. Any version pool page before that pointer (but after the "last" pointer) will never be needed by that read-only transaction.

2. Transaction Termination. When a read-only transaction terminates, its read range lower bound is compared with the reader_first pointer. If the two pointers are equal, the reader_first pointer is advanced to the minimum(6) of the read range lower bounds of remaining active read-only transactions. Similarly, when an update transaction terminates, its update range lower bound is compared to the update_first pointer. If the two pointers are equal, the update_first pointer is set to the minimum(7) of the update range lower bounds of the still active update transactions.

---

(5) The pool is full when (last - first) modulo VP_SIZE equals one.

(6) The computation of such minimum must account for the circular nature of the version pool.

(7) Again, this computation must account for the circular nature of the version pool.

3. A "free" version pool page is required. The version pool page that will be used is always "last" + 1. The last pointer is updated accordingly. Note also that each update transaction maintains a list of version pool pages that it has updated. In addition, the identity of that slot which is "closest" to the first pointer (i.e. the update range lower bound) is maintained.

When the version pool is not full, a read-only transaction never needs to be restarted. To understand why this mechanism works, suppose a read-only transaction T_r needs to read an old version of a page. T_r will read the latest version of the page, say version n, that was written by a transaction that was committed when T_r started. Let T_u be the update transaction that created the version after version n, (i.e., version n+1). Since T_r is not reading the version created by T_u, T_u could not have been committed when T_r was initiated. T_u was thus active when T_r was initiated or T_u was itself initiated after T_r. Therefore, T_u's update range lower bound must be greater than T_r's read range lower bound. Therefore, when T_u forced version n into the version pool, version n would be placed after T_r's read range lower bound.

Potential problems can arise if either the active read range or active write range includes the entire version pool. The system administrator can control whether potentially active slots are reused and whether new updates transactions are to be accepted by setting two parameters:

- Update Suppression Threshold (UST). When the number of pages not in the active update ranges falls below UST the system stops accepting new update transactions (although ongoing update transactions will be allowed to proceed).

- Abort Read-only Transactions Flag (ARTF). If this flag is set the reuse of potentially active slots is permitted. Either pages in the active read range, but not in the active update range, are reused — or the version pool lists of the update translations are used to find "free" pages within the active update range. In the former case, last is actually allowed to advance beyond reader_first. In the latter case, the update range lower bound for the requesting transaction is updated if necessary. Note that some read-only transactions may have to be aborted. If this flag is off, the update transaction that is requesting a slot from the version pool will be blocked until free slots become available.

As an alternative, we can dynamically allocate and deallocate additional version pools. In this case, we would maintain read-first and update-first pointers associated with each of the allocated version pools, and deallocate a "secondary" pool when no active transaction needs to make use of any of the pages in the pool. This fact can be determined in the course of updating the read-first and update-first pointers at the termination of transactions.

4. RECOVERY MANAGEMENT

The recovery management of the LDM is designed to support the smooth recovery from various kinds of systems and transaction failures. In particular, recovery from three types of failures is supported:

- Transaction recovery is performed when a user decides to abort an incomplete transaction or when the system backs out of a transaction in order to resolve a dead-lock situation.

- Soft crash recovery is performed when the DBMS comes back into operation after a software crash.

- Media recovery is performed when the information stored on disk is lost.

4.1 Transaction Recovery

To recover from a transaction failure is a very simple task in our scheme. We require that a list of all slots in the version pool that have been written by each transaction be maintained for the duration of the transaction. In the event of a failure, this list identifies the slots in the version pool containing before-images that will be used to undo the actions of the incomplete transaction. Since versions in the version pool are self-identifying, the undoing process is straightforward.

4.2 Soft Crash Recovery

Information stored in main memory is assumed to have been lost or corrupted after a software crash. Information stored on nonvolatile storage devices (disks), on the other hand, can be assumed to have remained intact. In many contemporary systems, it may be necessary to roll back the updates of incomplete transactions that have already been made to data stored on disk, and to redo the updates of transactions that have been committed but not yet reflected on disk (due to buffering by the system). In the LDM, we require that updates made by a transaction be forced out at transaction commit point. Therefore, no roll forward is necessary to recover from a "soft" system crash. To roll back the actions of incomplete transactions, the version pool is scanned. Each page version's header is examined. If the identifier of the transaction that has copied this version to the version pool is present in the completed transaction list, this page is ignored. If this page was written by an incomplete transaction, the version is simply copied back to the appropriate segment proper. Note that for the purpose of system recovery, an aborted transaction is treated just like a completed transaction.

When the version pool is large, it may be desirable to avoid having to scan the entire pool to look for "before images" written by incomplete transactions. Instead, it would be useful to delimit the set of slots in the version pool that may have to be examined. The portion that should be scanned should start from the page pointed to by the update-first pointer. The Overwriting Transaction Identifier associated with this first scanned page, call it OTIF, can be noted. The scanning can be stopped when a page is encountered with an Overwriting Transaction Identifier that is less than OTIF. Since the update-first pointer is recomputed at the end of each update transaction, it can be recorded with the modified completed transaction list without requiring an additional disk access.

4.3 Media Recovery

The standard approach to recover from this type of failure requires periodic dumping of the entire database to archival storage (like tape), and the maintenance of a transaction log to record all updates that have been performed since the last dump. To recover, the dumped version of the database is reloaded, and the updates performed by transactions beyond the time of the dump are redone.

In our scheme, the version pool is utilized to periodically dump the entire database. Unlike conventional DBMSs, in which a database must be taken off-line before a dump can be taken, the version pool is used to provide the capability of leaving the database on-line while dumping is in progress. The dumping process functions as an extremely long read-only transaction that reads every page in every segment and copies it to tape. Since updates are allowed to proceed, the dumping process will make use of the old versions in the version pool if necessary to obtain a consistent view of the database. A slight modification needs to be made to the algorithm that selects slots in the version pool for reuse. While the dumping process is going on, an old version cannot be released if it belongs to a segment that has not yet been dumped. When such a version is encountered, it is simply skipped over. This might have implied that the page had to be examined before allocation or that the version pool lists of ongoing update transactions had to be scanned. Again, an optimization is used to avoid such operations under most circumstances. The bounds of an active dump range are maintained (similar to the notions of active read range and active update range in Section 3.2). Slots that are not in any of the ranges are free and therefore can be reallocated without requiring an additional disk read. (Details of the modified algorithm can be found in [Chan81c].)

It is still necessary to maintain transaction log for recording after images. However, as hard crashes can be expected to occur rarely, we prefer to conserve storage space and at the same time reduce I/O overhead by recording the actions of transactions in terms of record level operations within individual pages, instead of storing the after images of individual pages. (Details of this logging scheme can be found in [Chan81b].)

5. SUMMARY

We have presented the implementation level design of an integrated concurrency control and recovery scheme based on the maintenance of multiple versions of data objects in a database. The garbage collection algorithm to be used with this scheme has been optimized to allow the efficient reuse of storage space occupied by page versions that will no longer be accessed. The concurrent execution of conflicting retrieval and update transactions and the simplified transaction backout in our integrated concurrency control and recovery scheme seem to have many advantages.

6. ACKNOWLEDGEMENTS

We would like to thank Professor Philip Bernstein and Professor Nathan Goodman for recommending to us the use of the multi-version concurrency control scheme described in this paper. A high level description of this integrated scheme has been obtained by Professor Bernstein through private communications with G. McLean who was with Prime Computer. The basic ideas behind this multi-version scheme are due to Christopher Earnest who is currently with Prime Computer.

7. REFERENCES

[Bayer80]
Bayer, R., H. Heller, A. Reiser, "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems, Vol. 5, No. 2, June, 1980.

[Chan81a]
Chan, A., S. Fox, K. Lin, D. Ries, "The Design of an Ada Compatible Local Database Manager", Computer Corporation of America. 1981 (submitted for publication).

[Chan81b]
Chan, A., S. Fox, A. Nori, D. Ries, "Local Database Manager (LDM) Subcomponent Architecture: Core Database Handler", Computer Corporation of America. 1981.

[Chan81c]
Chan, A., S. Fox, A. Nori, D. Ries, "Local Database Manager (LDM) Technical Documentation: Physical Database Processor", Computer Corporation of America, in preparation.

[DoD80]
United States Department of Defense, "Reference Manual for the Ada Programming Language", Proposed Standard Document, July 1980.

[DuBourdieu82]
DuBourdieu, D. J., "Implementation of Distributed Transactions", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, 1982.

[Eswaran76]
Eswaran, K. P., J. N. Gray, R. A. Lorie, I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November, 76.

[Gray78]
Gray, J., "Notes on Database Operating Systems", Operating Systems -- An Advanced Course, R. Bayer, R. M. Graham, and G. Seegmuller (editors), Springer Verlag, New York, 1978.

[McLean80]
McLean, G., private communications.

[Reed78]
Reed, D. P., "Naming and Synchronization in a Decentralized Computer System", Ph.D. dissertation, Department of Electrical Engineering and Computer Science. M.I.T., Cambridge. Massachusetts, September, 1978.

[Shipman81]
Shipman, D., "The Functional Data Model and the Data Language DAPLEX", ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[Stearns81]
Stearns, R., D. Rosenkrantz, "Distributed Database Concurrency Controls Using Before-values", ACM SIGMOD Conference Proceedings, 1981.